(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A GRAPHICAL EDITOR FOR THE COMPUTER AIDED PROTOTYPING SYSTEM

by

Roger K. Thorstenson

December 1988

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A GRAPHICAL EDITOR FOR THE COMPUTER AIDED PROTOTYPING SYSTEM

12. PERSONAL AUTHOR(S)
Thorstenson, Roger, K.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	December 1988	139

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
rapid prototyping, software development, real-time, graphical editor, user interface, abstraction, prototyping environments

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Computer Aided Prototyping System (CAPS) is used to develop executable prototypes of large embedded software systems with hard real-time requirements. The system is based upon the Prototype System Description Language (PSDL) and a set of integrated software development tools. A graphical tool is needed to decompose the PSDL composite operator into a network which shows its component parts and their communication paths. The graphical description is part of the specification of the intended system's components.

This thesis explores the requirements of such an editor and demonstrates that it can be developed. It shows that the editor can effectively be used to decompose operators and that it can produce an equivalent textual representation in PSDL.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
XX UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Lugi	(408) 646-2735	52Lq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

☆ U.S. Government Printing Office: 1986—606-24

Approved for public release; distribution is unlimited

A Graphical Editor for the Computer Aided Prototyping System

by

Roger K. Thorstenson
Lieutenant, United States Navy
B.S., University of Idaho, 1982

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

Author: _____
Roger K. Thorstenson

Approved by: _____
Luqi, Thesis Advisor

_____
Michael Zyda, Second Reader

_____
Robert B. McGhee, Department of Computer Science

_____
Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

The Computer Aided Prototyping System (CAPS) is used to develop executable prototypes of large embedded software systems with hard real-time requirements. The system is based upon the Prototype System Description Language (PSDL) and a set of integrated software development tools. A graphical tool is needed to decompose the PSDL composite operator into a network which shows its component parts and their communication paths. The graphical description is part of the specification of the intended system's components.

This thesis explores the requirements of such an editor and demonstrates that it can be developed. It shows that the editor can effectively be used to decompose operators and that it can produce an equivalent textual representation in PSDL.

TABLE OF CONTENTS

ACKNOWLEDGMENT

My gratitude goes to my thesis advisor, Professor Luqi, for her guidance and encouragement throughout the months of preparation and development of the Graphical Editor. I would also like to thank the following people, who worked on other aspects of the Computer Aided Prototyping System (CAPS), for their esprit de corps and clarification of their portions of CAPS:

LT Charlie Altizer

LCDR Dan Galik

LCDR Laura Marlowe

LT Scott Porter

CAPT Hank Raum

LT Mary Lou Wood

Special thanks to LCDR John Yurchak for guidance and expertise necessary for this research.

Finally, I wish to thank my wife, Georgia, and daughter Kari, for without their encouragement and love, this thesis would not have been completed.

I. INTRODUCTION

A. RAPID PROTOTYPING

A need to improve the productivity of software development and the reliability of developed programs has resulted in research aimed at developing a system which allows a designer to rapidly generate prototypes of large real-time software systems. This system is based on replacing the traditional software life cycle with a two phase cycle consisting of rapid prototyping and automatic program generation. [Ref. 1]

The process of designing a large real-time software system is long and labor intensive if it is feasible at all. Determining whether a system is feasible is often something which must be checked and rechecked as the design of a system progresses [Ref. 2]. Determining that a very large system is unfeasible after the entire system has been developed results in a tremendous loss of time and money. A computer-aided prototyping system (CAPS) to solve this problem is being studied. The rapid prototyping technique provides the designer with a means of writing specifications, based on user requirements, for components of the system being designed and then using reusable software components which match the specifications to build a prototype of the intended system. This prototype can then be tested to see if the design meets the user's needs or whether the user's

requirements are even feasible [Ref. 3]. The result is a much more timely and cost effective answer to the questions of feasibility and whether the design meets the specifications.

B. CAPS OVERVIEW

The CAPS prototyping environment is comprised of a specification language and a set of integrated prototype development tools. The Prototype System Description Language (PSDL) is used to describe connections between components of the system being prototyped and to specify the behavior of the reusable components used in constructing the prototype. PSDL provides the designer with a means of creating and assembling abstractions of the components of the system being designed. The prototype development tools in CAPS include a user interface with sequence control function, syntax directed editor, graphical editor, design database, database of reusable software components, rewrite system and an execution support system. With these tools and the PSDL language the designer can develop an executable prototype of the system he is designing. This prototype will not actually be the intended system, but rather will be a partial representation of the system which can be used to analyze various aspects of the design. Since the prototype design is based on system requirements, it can be used to demonstrate their adequacy. Adjustments to the prototype can be made based on customer feedback. It can also be used to validate

attributes, timing constraints, I/O formats and module interfaces.

C. GRAPHICAL EDITOR

In order to construct a prototype of a large real-time software system, the designer needs tools which help him specify and decompose the system. In the CAPS system, a syntax directed editor is used to enter the specifications of the components of the system being designed. Based on this specification, CAPS tries to find a reusable module which matches (or nearly matches) the specification. If the search is unsuccessful, a method of decomposing the component is needed. The tool discussed in this paper is a graphical editor which the designer can use to decompose components of the system being designed into their subcomponents. This tool provides the user with a visual representation of the component decomposition and an equivalent PSDL representation which is used by CAPS to aid in the construction of the prototype. [Ref. 3]

Chapter 2 of this paper discusses the CAPS environment in terms of the individual tools used for prototype development and their functional connection to the graphical editor. Chapter 3 provides a detailed description of how the graphical editor tool is used for both initial system decomposition and for editing an existing prototype. It also discusses critical issues involved in the decomposition process. The fourth chapter provides a detailed description

of the design of the graphical editor. Chapter 5 contains the conclusions and discusses possible follow on research.

II. THE CAPS ENVIRONMENT

A. THE PROTOTYPING LANGUAGE

1. PSDL Background

The high level programming language Ada has several features which support the development of reusable modules. These features include information hiding, data abstraction, and generic packages, among others [Ref. 4].

PSDL is a prototyping language designed specifically for describing real-time software systems. It supports the rapid prototyping concept which is based on abstractions and piecing together of reusable components. There is a high degree of compatibility between Ada and PSDL, because of their similar features and the similar philosophies of large system design upon which they are based. In fact, the execution support system of CAPS which produces an Ada representation of the design is itself written in Ada.

A prototype developed using CAPS is an executable modularized skeleton of the intended system which is used to validate the critical aspects of the system. PSDL is used to describe the behavior of components and their interconnections. Its design was motivated by the need for several capabilities including:

1) support of modularized design with explicit module interconnection.

2) an executable prototype.

3) the ability to decompose large complex systems in a simple way.

4) a common notation for specification and design.

5) a way to specify reusable module retrieval.

6) both formal and informal methods of specification so that the designer could choose the method most appropriate.

7) support for data abstraction, functional abstraction and control abstraction.

8) a set of built in abstract types for use in constructing real-time systems.

Since no existing language satisfied all of these requirements, PSDL was developed. [Ref. 5]

2. PSDL Constructs

PSDL is designed to provide the designer with a means of creating a model of a real-time system. With PSDL, the designer can hierarchically decompose systems based on data flow and control flow. PSDL provides mechanisms for creating abstractions for operators, data and control.

a. PSDL Model

The mathematical model behind PSDL is an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the maximum execution time associated with each vertex v, and C(v) is the set of control constraints associated with each vertex v. The vertices represent operators and the edges represent data streams. The first three components of

this model are represented by the data flow diagram which the designer develops when he is decomposing an operator with the graphical editor. [Ref. 5]

(1) Operators. Operators are used to represent the components of the prototype. They can be used to represent functions or state machines. Communication between operators is carried out via data streams. These data streams carry values of a fixed abstract data type or values of the PSDL built in type EXCEPTION. An operator can be either data driven or periodic. Periodic operators fire at some interval whereas data driven operators fire in response to some stimulus from an input. When an operator fires, two things occur. First, it reads one data object from each of its input streams. It then writes at most one data object on each of its output streams. When a function operator fires, output objects are produced whose values depend upon the current set of input values. State machines produce outputs which depend upon the current set of inputs and the current values of their internal state variables. Operators can also be characterized as being either atomic or composite. If an operator cannot be further decomposed into more detailed parts, it is atomic. A composite operator can be further decomposed into data and control flow networks of lower level operators. A composite operator whose decomposition contains a cycle is a state machine. [Ref. 5]

(2) Data Streams. Data streams are the communication links between two operators and as such carry a sequence of data values. The data streams will be one of two types. The first type is called a DATA FLOW STREAM and is used to represent a FIFO queue. This type data stream guarantees that none of the values it carries will be lost or replicated. It also restricts the relative firing rates of the two associated operators. Data flow streams are used where each data value is unique and must be acted on exactly one time. The second type of data stream is called a SAMPLED STREAM. This type of data stream makes no guarantees as to whether its values will be lost. It is a cell capable of carrying just one value which is updated whenever the producer operator generates a new value. It can deliver the same value more than once if requested and will discard an unused value when a new one is generated. Sampled streams are used to simulate continuous streams of information where only the most recent values are meaningful. Whether a data stream represents a data flow stream or sampled stream is determined by the consumer operator's activation conditions. [Ref. 5]

(3) Exceptions. PSDL has a built in abstract data type called EXCEPTION. Values of this type are transmitted along data streams like values of a normal type. The exception type has operations which create named exceptions, detect whether the value is a particular named

exception or detect whether the value is of a type other than exception. [Ref. 5]

b. Abstractions

PSDL provides the designer with operator, data and control abstractions as a means of controlling the complexity of his design.

(1) Operator Abstraction. Operators consist of a specification and an implementation. The specification contains the form of the interface, timing constraints and a formal and informal description of the operator's observable behavior. The implementation part determines whether the operator is atomic or composite. An atomic operator has a keyword ADA, which specifies Ada as the implementation language. This is followed by the implementation module for the operator which is the result of a successful retrieval of a reusable software component, or the writing of the Ada component. An operator which can be further decomposed into component operators is called a composite operator. Its implementation will be a series of PSDL statements which show the relationships between its components. [Ref. 5]

There are two kinds of operator abstractions: functional abstraction and state machine abstraction. The difference between the two is that a state machine abstraction has a state declaration in its specification part, whereas a functional abstraction does not. [Ref. 5]

(2) Data Abstraction. Data abstraction is used to decouple a data type's behavior from its representation. It allows interfaces to be described independent of their data representation. Therefore, interfaces in the prototype are the same as those in the system being designed. This simplifies system validation and enables the structure of the prototype to be used in the intended system. [Ref. 5]

All PSDL data types are immutable, meaning that they cannot communicate via side effects. The data types include: those Ada built in types which are immutable, user defined abstract data types, TIME, EXCEPTION and the types which can be built using the PSDL type constructors listed in Table 1. [Ref. 5]

TABLE 1. PSDL TYPE CONSTRUCTORS [Ref. 5]

```
set[item:type]
sequence[item:type]
map[from:type,to:type]
tuple[tag_1:T_1,...,tag_n:T_n]
one_of[tag_1:T_1,...,tag_n:T_n]
relation[tag_1:T_1,...,tag_n:T_n]
```

(3) Control Abstraction. Control abstraction in PSDL is represented by the enhanced data flow diagrams developed with the graphical editor together with a set of control constraints. The data flow relationships of the enhanced data flow diagram determine the order of execution of the operators. [Ref. 5]

c. Control Constraints

An operator's control is specified by its control constraints. The aspects of the operator which are specified by its control constraints are: whether it is PERIODIC or SPORADIC, its triggering condition and its output guards. If an output guard is not satisfied it prevents outputs from being placed on the output streams. Periodic operators are triggered by the static scheduler at some specified interval whereas sporadic operators trigger on the arrival of new data.

An operator's triggering condition can be either BY ALL or BY SOME. If the triggering condition is BY ALL, the operator will fire only when it has new values on all of its inputs. This ensures that its output is based on the presence of fresh data on all of its inputs. The BY ALL triggering condition is used to synchronize processing when there are inputs from many data streams. If the BY SOME triggering condition is used, the operator will fire when any of its inputs get a new value. This guarantees that the operator's output is based upon the most recent input values. [Ref. 5]

All operators are required to have a firing period or a data trigger and may have both. Periodic operators which have data triggers are conditionally executed with the data trigger serving as the input guard.

A timer is a abstract state machine used in keeping track of time between events, or the amount of time spent in a state. It is important that a system for modeling real-time systems has a facility for expressing event timing. There are four operations which can be performed on timers: START, STOP, READ and RESET. These are analogous to the operations performed on a stop watch. [Ref. 5]

Two kinds of conditionals are available in PSDL: conditional execution of an operator and conditional transmission of output.

A triggering condition acts as a guard for an operator. If an operator's triggering condition is true, it reads its inputs and fires. If the triggering condition is not true, the operator will read its input data streams but will not fire. The triggering conditions for periodic operators are tested at an interval specified by its period. [Ref. 5]

An output guard gives the effect of passing an unconditional output through a filtering operator. If the triggering condition is satisfied, the filtering operator fires and passes its input values on. In the event that the triggering condition is not met, the filter removes the value from its input stream without effecting its output stream. [Ref. 5]

d. Timing Constraints

Since timing is a critical issue in real-time systems, any language used to specify these systems must have provisions for expressing the timing aspects of a system's operation.

There are three basic timing constraints in PSDL: MAXIMUM EXECUTION TIME (MET), MAXIMUM RESPONSE TIME (MRT) and MINIMUM CALLING PERIOD (MCP). An operator's MET is the maximum amount of time it takes to execute from the instant it starts to the instant it completes. A sporadic operator's MRT is the maximum amount of time between the arrival of new data and the time the last value is put into an output stream in response to the new input value's arrival. In the case of triggered BY ALL, "new data" means the last new input value in the ALL list. For triggered BY SOME, "new data" means any new input value in the SOME list. The MRT for a periodic operator is the maximum amount of time from the beginning of its firing period to the time when its last value is put onto an output stream during that period.[Ref. 5]

The MCP applies only to sporadic operators. It specifies the least amount of time between the arrivals of sets of input values. Every sporadic operator with an MRT must have an MCP. The MCP must be greater than or equal to the MRT for state machine operators in either single or multi-processor cases and for function operators in the single processor case only.

More complex timing constraints can be given by using event controlled timers, triggering conditions and output conditions.[Ref. 5]

e. Hierarchical Constraints

Any PSDL operators which are not atomic must be decomposed into their component operators. This decomposition is done hierarchically from top down. There are several constraints associated with the hierarchical structure. Inputs and outputs which occur at one level of decomposition must occur at some point in each subsequent level of decomposition. Data streams which are introduced by decomposing an operator into a network of operators must also appear in each further level of decomposition as inputs and outputs. METs and MRTs within a decomposition can be no longer than those of their associated composite operator. An operator's decomposition inherits its period, MCP, and the types of its input and output data streams. A composite operator inherits the exceptions produced by its component operators.[Ref. 5]

B. PROTOTYPE DEVELOPMENT TOOLS

The development of a prototype with CAPS progresses in much the same way as a programmer develops a program. The primary difference being that with CAPS the user is provided with a means of designing at a level of abstraction above that of even a high level programming language. Another difference is that with CAPS the design is driven directly by

the user's specifications which ensure that the end product truly will meet the requirements of the intended system.

A typical programming environment for creating programs provides an integrated set of tools which support program development and testing. All have a user interface of some type (be it friendly or unfriendly), some form of textual or graphical editor for creating and modifying the program, some means of storing the program, and a way of testing and debugging the program. Program development progresses as a cycle of entering a program, repeatedly testing, debugging and modifying it until it runs as desired.

The CAPS prototyping environment has tools which perform tasks analogous to those of programming environments. One of the primary differences between CAPS and a programming environment is that a CAPS prototype is based directly upon system requirements. Another difference is that CAPS, unlike most other environments, is heavily dependent upon the existence of reusable ADA software components. CAPS has tools which support storage of reusable software components, formally describing reusable software components, initially developing the prototype and support for the test, debug and modify phase of development.

1. **System Control**

CAPS is controlled by the sequence control function of the user interface subsystem [Ref. 6]. This portion of the system provides the user with an easy method of invoking

tools, hides the details of tool interaction and prevents the user from making system operation errors.

2. Prototype Creation and Modification

Two tools are available which support creation and modification of the intended prototype. The syntax directed editor provides the designer with a means of entering PSDL specifications and control constraints for the various system components. The graphical editor provides a means of decomposing composite operators into networks of component operators connected by data streams. When the designer signals to the graphical editor that the decomposition is complete, the editor scans the operators and data streams and creates PSDL link statements [Ref. 7] which are textual representations of the graphical objects.

The syntax directed editor and graphical editor are used together in an iterative fashion to enter the specifications of the components of the system, specify their connections and timing relationships and to ultimately decompose the entire system into its atomic parts. Each atomic operator will either be implemented by an existing reusable Ada module or must be coded.

As the decomposition process takes place, a prototype of the intended system will be assembled in the design database. Once the entire specification and decomposition process has been completed, the design data base will contain an abstract tree representation of the entire system. The

central nodes of the tree will contain the information for the composite operators and the leaves will be the atomic operators. The design database has facilities for retrieval, addition and deletion of the PSDL objects that it stores.

3. Prototype Testing Tools

The execution support system (ESS) of CAPS is used to test the designer's prototype, which is stored in the design database, discover errors and make reports to the user so that he can make the necessary design modifications. The components of the ESS are: a TRANSLATOR, a STATIC SCHEDULER and a DYNAMIC SCHEDULER.

The translator's function is to convert each PSDL specification into an executable Ada module [Ref. 8]. The static scheduler looks for operators which have timing constraints which must be met. It then creates a schedule which specifies the order in which the time critical operators must execute. The translated modules and the static schedule are passed on to the dynamic scheduler which executes the translated modules in the order specified by the static schedule. If a time critical module runs in less time than was allocated by the static scheduler, the dynamic scheduler will schedule non-time critical operators (modules) to run during the time gap. The dynamic scheduler also provides feedback to the user interface based on run time exceptions generated by the translator or static scheduler.

4. Reusable Component Support Tools

The rewrite subsystem of CAPS is used to normalize the specification of Ada modules. The reusable software component database is used to store reusable Ada components. It has facilities for storing, searching for and retrieving these reusable components based on their normalized specification.

C. SYSTEM OPERATION

Development of a prototype using CAPS progresses in two distinct phases:

1) Initial System Design
2) Test and Modify

1. Initial System Design

During this stage of development of the prototype, the editors are used to construct an abstract tree representation of the prototype.

Initially, the designer uses the syntax directed editor to enter the specification part of the highest level operator. The sequence control function then initiates a search of the software database for a component which matches the normalized description of this specification. This search will yield one of three possible results. The first possibility, which will become more likely as the software database matures, would be that a reusable component is found which matches the specification. In this case, the reusable component is retrieved and is attached to the specification

as the implementation part. The initial design stage would be complete at this stage.

The second possibility would be that a reusable module could not be found, but the operator is atomic. Atomic operators are those for which no further decomposition is necessary. In this case, the implementation part simply contains the keyword ADA signifying that the implementation must be coded. Again, the design stage would be complete.

The third and most likely possibility, especially at high levels in the design, is that a reusable component will not be found and the operator is composite. In this case, the graphical editor must be used to decompose the operator into a network of operators and data streams at the next level of detail. Once the decomposition is complete at this level, the graphical editor will transform the graph into PSDL link statements. These link statements become part of the implementation part of the operator's specification. Next, the syntax directed editor is used to enter any control constraints which are needed to complete the specification of the operator's behavior. Once the entire specification is complete, the sequence control function notifies the design database of the existence of the new operators so that it can allocate their storage. Finally, the PSDL specification file for the decomposed operator is stored in the design database as an object. The graphical information necessary to reconstruct the drawing is also stored with the object.

The process described is repeated until the intended prototype is completely comprised of atomic operators which are either reusable components or modules which must be coded.

2. **Test and Modify Phase**

Once the initial prototype design has been completed, it is necessary to test it and make necessary modifications. Before the testing can occur, the sequence control function must collect the specifications together and pass them to the ESS [Ref. 6].

a. **Testing the Design**

Before the prototype design can actually be executed, the PSDL specifications must be translated into Ada and the order of their execution must be determined. The translator scans through the PSDL specification file and translates each operator's specification part, implementation part and constraint part into an equivalent Ada module [Ref. 9]. If translation errors are found, translation exceptions will be raised, otherwise the translated modules are passed to the dynamic scheduler for eventual execution.

In order to actually execute the Ada modules, a schedule is required which will determine the order that the time critical modules will execute. The static schedule, in simple terms, is a series of calls to the translated modules which have time constraints [Ref. 10]. If scheduling errors occur then scheduling exceptions are raised. If no

scheduling errors occur, the schedule is passed on to the dynamic scheduler.

The dynamic scheduler actually performs the run time testing of the prototype. It uses the static schedule to call the translated modules. If one of the time critical modules does not use up its allocated time slot, the dynamic scheduler schedules non-time critical modules to run. If run time errors occur, run time exceptions are raised. Otherwise, the prototype is feasible and is known to meet the specifications.

b. Modifying the Design

Exceptions raised during translation, creation of the static schedule or during execution signal that the prototype design must be modified. The sequence control function of the user interface responds by invoking either the syntax-directed editor or graphical editor, as appropriate, so that the necessary corrections can be made to the design.

III. USING THE GRAPHICAL EDITOR

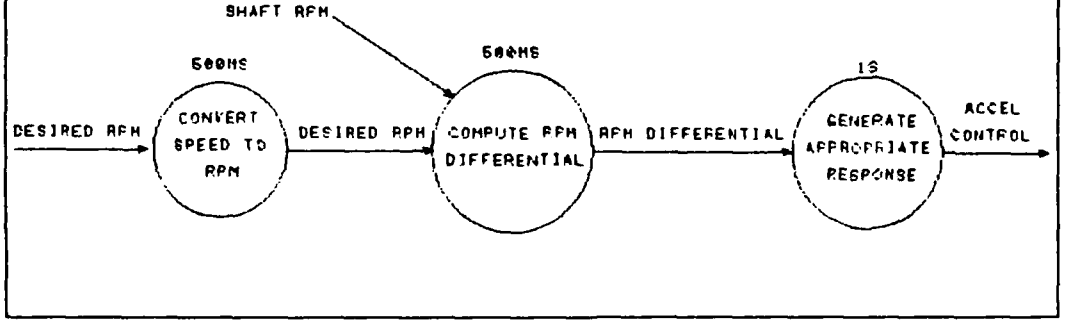
A. PURPOSE

The graphical editor is one of the designer's primary tools for designing the intended system. It provides him with a simple way of expressing what the parts of the system will be and how those parts interact in terms of time relationships and communications. Once the highest level operator has been entered, the graphical editor becomes the initial point of entry for all lower level operator names, data stream names and maximum execution times.

In order for the graphical editor to provide the designer with the maximum expressive power, it must support any type of graph which could possibly be used to represent a system's components and their associated data streams. It must then be able to convert the graphical representation of the graph into a textual form which captures the meaning intended by the picture. The PSDL link statements are used to express what data stream is used to communicate between two operators. Figure 1.(a) shows a PSDL specification of an operator named SPEED_CONTROL_SYSTEM. The specification part shows that the operator has inputs SHAFT_RPM and DESIRED_RPM and an output ACCEL_CONTROL. The two exceptions PRESENT_SPEED_TOO_SLOW and DESIRED_SPEED_TOO_SLOW are special conditions which will be handled external to the operator being decomposed. The maximum execution time shows that the

```
SPECIFICATION
   OPERATOR  SPEED_CONTROL_SYSTEM
      SPECIFICATION
         INPUTS  SHAFT_RPM     : real,
                 DESIRED_MPH   : real
         OUTPUTS ACCEL_CONTROL: controlling_action
         EXCEPTIONS PRESENT_SPEED_TOO_SLOW,
                    DESIRED_SPEED_TOO_SLOW
         MAXIMUM EXECUTION TIME 2s
         DESCRIPTION
         { Returns command to accelerate, decel-
           erate or hold present setting based on
           a comparison of the present speed with
           the desired speed. If the present speed
           or desired speed are below 30 mph
           exceptions are produced.
         }
      END
```

(a) Operator Specification



(b) Operator Decomposition

```
DESIRED_MPH.EXTERNAL-->CONV_MPH_TO_RPM
DESIRED_RPM.CONV_MPH_TO_RPM:500MS-->COMPUTE_RPM_DIFF
SHAFT_RPM.EXTERNAL-->COMPUTE_RPM_DIFF
RPM_DIFF.COMPUTE_RPM_DIFF:500MS-->GENERATE_RESPONSE
ACCEL_CONTROL.GENERATE_RESPONSE:1S-->EXTERNAL
```

(c) PSDL Link Statements

Figure 1 Decomposition of an Operator

operator must execute in 2 seconds. Therefore, no path in the operator's decomposition can have a total time greater than 2 seconds. Figure 1.(b) shows a graph of the operator's decomposition. It shows that SPEED_CONTROL_SYSTEM was decomposed into three operators: CONV_MPH_TO_RPM, COMPUTE_RPM_DIFF and GENERATE_RESPONSE. Each operator's MET is shown above it. DESIRED_MPH and SHAFT_RPM are shown as inputs to CONV_MPH_TO_RPM and COMPUTE_RPM_DIFF respectively. ACCEL_CONTROL is shown as an output from GENERATE_RESPONSE. The lines labeled DESIRED_RPM and RPM_DIFF are data streams. After the decomposition has been completed, the syntax directed editor will be used to enter the control constraint part of the PSDL specification. This is used to more precisely specify the operator's behavior.

Figure 1.(c) shows the PSDL link statements which the graphical editor would produce for the graph shown in Figure 1.(b). PSDL link statements have the syntactic form

```
data stream . source [:MET] --> destination
```

where :MET is optional. The maximum execution time must be an integer value and have units as shown in Table 2.

TABLE 2 PSDL TIMING UNITS

UNITS	MEANING
m	minutes
s	seconds
ms	milliseconds

The source operator name and destination operator name is "EXTERNAL" for data streams which come from or go to an operator external to the current diagram.

In order for the designer to be able to alter an existing graph in response to errors found during testing, the graphical editor must have the capability to reload and redraw a previously decomposed operator's diagram. After this has been done, the designer can edit the diagram by adding and deleting objects from it.

B. DECOMPOSING AN OPERATOR

When it has been determined that an operator must be decomposed into its components, the user will start the graphical editor. In order to reduce the amount of information that the designer must memorize, the following information must be retrieved from the operator's specification part and be made available:

1) OPERATOR'S NAME
2) LIST OF INPUTS TO THE OPERATOR
3) LIST OF OUTPUTS FROM THE OPERATOR
4) LIST OF STATES
5) THE OPERATOR'S MAXIMUM EXECUTION TIME

Making this information available to the designer helps to ensure correctness between levels of the graph, because once a data stream is identified on one level, it will exist on all subsequent levels of that operator's decomposition. Once a MET is assigned, each level of the operator's subtree

must execute in that same amount time or less. This is where the hierarchical constraints of PSDL are realized.

1. Using the Graphical Editor

The graphical editor runs in a window environment. It has five editing modes which allow the user to draw five types of objects: operators, external inputs, external outputs, data streams and self loops (PSDL states). The default editing mode is DRAW OPERATOR. In this mode the designer can draw expandable bubbles with names and time constraints (See Figure 2.(a)). The DRAW DATA STREAM mode allows lines to be drawn between two previously drawn operators (See Figure 2.(b)). Input lines coming from some source external to the operator being decomposed can be drawn in the DRAW INPUT mode (See Figure 2.(c)). Output lines are drawn in the DRAW OUTPUT mode. These lines designate data streams which leave the operator being decomposed (See Figure 2.(d)). The DRAW SELF LOOP mode allows the designer to draw state variables (See Figure 2.(e)). Syntax checks are done to ensure that the graphical objects are drawn in a manner which will ensure that the resulting PSDL link statements will be correctly formed. In addition to syntax checking the graphical editor also does some semantic checking. These checks try to ensure that what the designer draws conveys its intended meaning and also that the resulting link statements convey the same meaning as the diagram.

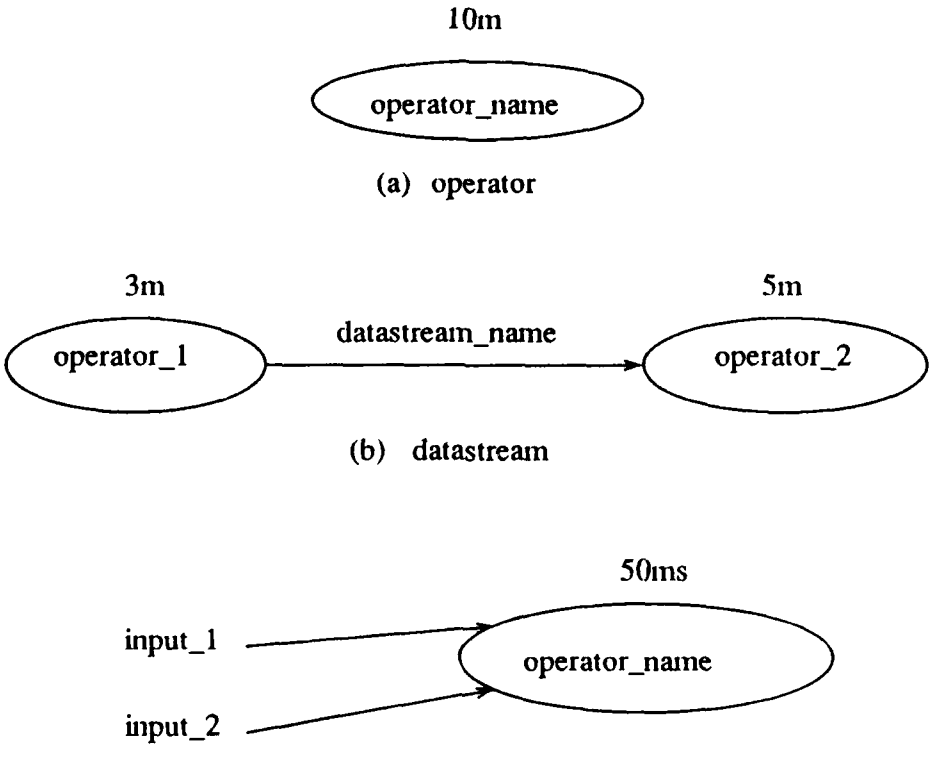


Figure 2 Graphical Objects Used to Decompose Operators

C. CORRECTNESS AND CONSISTENCY

1. Syntactic Correctness

Drawing graphical representations of networks of operators is a high level form of visual programming. As with any language, this pictorial language has an associated syntax. The graphical editor provides a medium for constructing these visual programs. It will only accept symbols which it knows are in the visual language that it understands. User input which fails to fit the syntax of the pictorial language is ignored.

The operator decomposition language consists of expandable bubbles and arrows. The bubbles which are used to represent operators can be drawn in any size, but must not overlap other operators. Allowing bubbles to overlap would cause the problem of ambiguous "picking" when connecting or deleting operators.

Arrows represent data streams. Since an arrow can have four different meanings in the language, its syntax is more complex than operators. Arrows which represent inputs to the decomposition diagram, must start on a point in the drawing space not occupied by an operator and must terminate on an operator. Arrows representing output lines must originate on an operator and terminate on a spot in the drawing area which is not occupied by an operator. Data streams are represented by arrows which start on one operator

and terminate on another. PSDL state variables are drawn using the self loop mode of the editor. A self loop's arrow must start and terminate on the same operator. Lines which are not syntactically correct are rejected by the editor and immediately erased.

A correctly formed PSDL operator decomposition diagram has all of its component objects named. Therefore, the graphical editor will not allow an object to be drawn which has not been named. In the case of operators, there must also be a MET entered. All names used in the decomposition will eventually be used as either module names or names of abstract data types (ADT) in the intended prototype. Since these modules and ADTs will be implemented in Ada, the graphical editor requires that all object names are syntactically correct Ada identifiers (See Figure 3). The graphical editor also checks that an operator's MET is syntactically correct (See Figure 4).

```
identifier ::= letter{[underline]letter_or_digit}
letter_or_digit ::= letter | digit
letter := upper_case_letters | lower_case_letters
```

Figure 3 Backus-Naur Form of Ada an Identifier [Ref. 11]

```
MET ::= <digit_string> <time_unit>
digit_string ::= digit | digit_string
time_unit ::= m | s | ms
```

Figure 4 Backus-Naur Form of Maximum Execution Time

2. Semantic Correctness

The same mechanisms used to do syntactic checking of a decomposition also provide the semantic checking. The editing mode tells the editor what the designer's drawing actions should mean. If something is drawn which does not fit the syntactic criteria for the object indicated by the mode, the system rejects it. This ensures that the decomposition will have the designer's intended meaning. Without this comparison of editing mode and specific object criteria the graphical editor would have no way of ensuring that the designer's actions actually conveyed their intended result.

3. View Consistency

CAPS is faced with a problem common to many software development environments which represent objects in multiple ways. That is, how to ensure that all other views of an entity are appropriately changed when one of them is altered.[Ref. 12].

A view is a way of representing an object within a system. Each decomposed operator has a graphical representation which is used to reconstruct the picture and a textual PSDL specification representation. After an operator is completely decomposed with the graphical editor, the editor automatically performs a transformation on the picture and produces the equivalent PSDL link statement representation. From this point on, the problem of

maintaining view consistency exists. A change in one view of the object should automatically result in a corresponding change to the object's other view [Ref. 12]. Failure to keep the PSDL specification view and the graphical view consistent will rapidly destroy the integrity and intelligibility of the design. Since CAPS does not currently have a method of running concurrent view editors, some other method must be used to ensure view consistency.

One way of ensuring view consistency is to make the requirement that changes to the specification must be made in the graphical editor instead of the syntax-directed editor if the change involves any of the following items:

1) operator names
2) data flow names
3) connections between operators
4) maximum execution times
5) adding new objects to a decomposition
6) deleting objects from a decomposition

For changes involving these items, the existing decomposition should be reloaded into the graphical editor, the changes should be made as needed and new PSDL link statements and a new diagram file should be generated. Then the syntax directed editor should be used to make any further changes to the operator's specification which are necessary. Changes to the specification which do not involve the six listed items

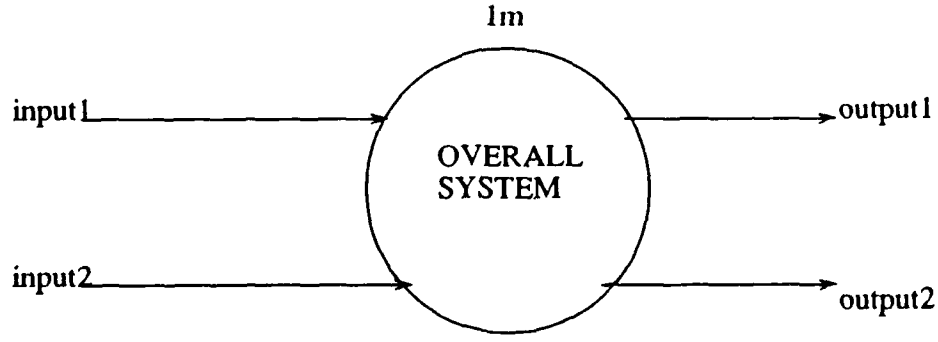
could not cause this problem and therefore do not require this restriction.

4. Consistency Between Levels of Decomposition

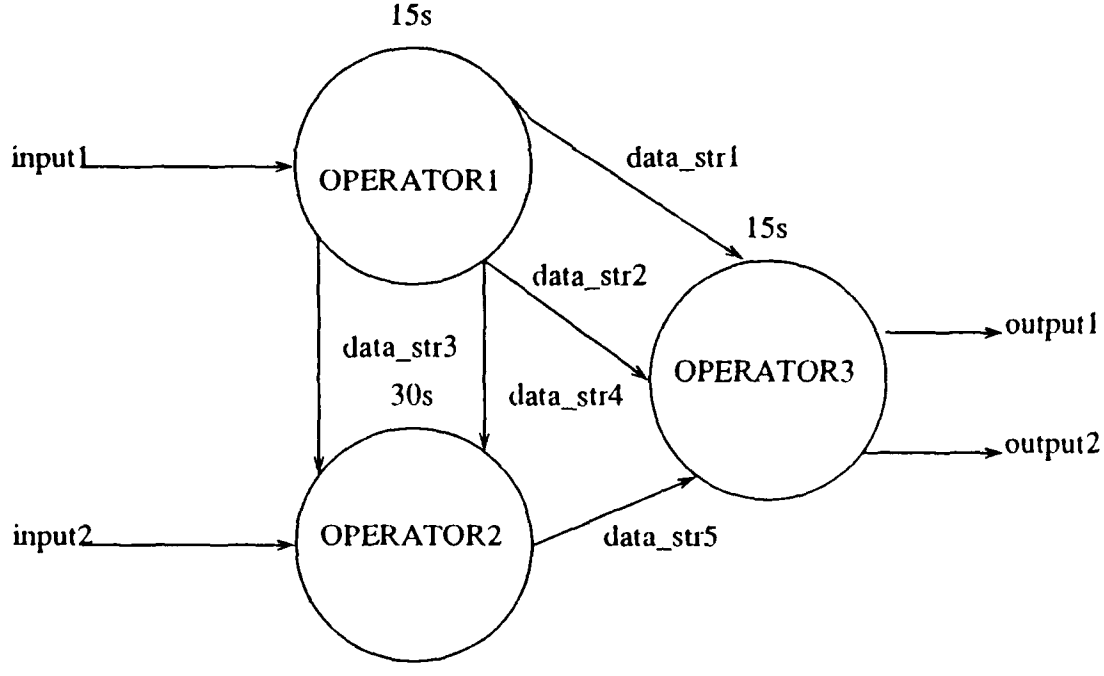
After an operator has been decomposed, a new node is created in the design database for each new component operator. Each input, output and state of the operator being decomposed must be represented as an input or output to some operator in its decomposition (See Figure 5).

Ensuring consistency between hierarchical levels during the initial decomposition poses very little problem because the sequence control function creates a specification shell for each new operator in the decomposition [Ref. 6]. This shell contains a portion of each new operator's specification part (See Figure 6). When the graphical editor is used to further decompose each of these operators, the sequence control function should pass in the operator's name, its input list, output list, state list and MET. Making these items available will help the designer ensure that each input, output and state of the operator being decomposed is used in the decomposition. In addition, the availability of the MET will reduce the likelihood of a time constraint violation in his decomposition.

Maintaining consistency between levels when making modifications is more difficult for certain types of changes. If an operator in any particular level is deleted, his entire subtree must be deleted from the design database. This is a



(a)



(b)

Figure 5 Decomposition of Operator into Its Component Operators

Figure 6 Stub Specifications Resulting From Operator Decomposition

simple process. However, if an operator's connectivity is changed by deletion, addition or by redirecting of data streams, its entire subtree must be rechecked to ensure consistency. This should be done in the same top down fashion as decomposition, having the sequence control function pass in the input, output and state names for the appropriate operator as is done with decomposition. Also of value in this process is the ability to look up or down a level in the decomposition hierarchy. However, allowing the designer to randomly edit up and down between levels could cause inconsistencies between the pictorial view and the specification. The problem being that as the diagrams are changed, the specifications would remain unchanged. With a sufficient number of changes, it could become very difficult for the designer to make the textual and pictorial views match.

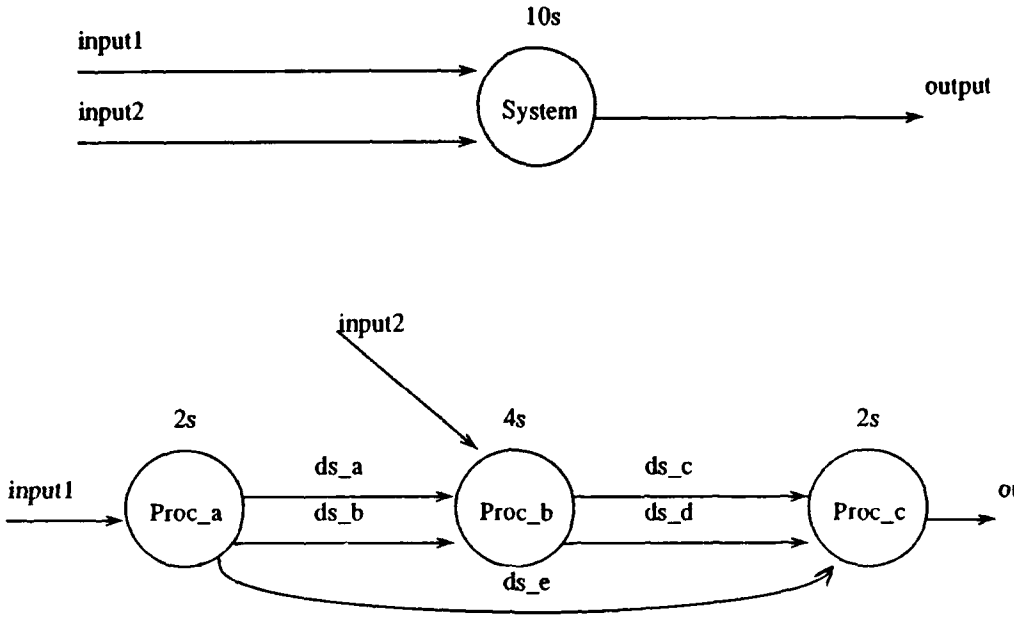
IV. DESIGN OF THE GRAPHICAL EDITOR

A. STATIC SCHEDULER CONSIDERATION

The static scheduler is the portion of CAPS which uses the decomposition information contained in the PSDL link statements. Therefore, it has a major impact on the design of the graphical editor.

The static scheduler determines precedence relationships between operators by analyzing the link statements. Research was conducted with the intent of reducing the amount of time and memory required to perform the scheduling task. One possible solution, which was investigated, was to supply the scheduler with the exact source and destination of the inputs and outputs of the decomposition. *Doing this* would require the static scheduler to only consider the link statements of the composite operators which are parents of atomic operators. For a large system this would be a large reduction in the quantity of operators considered.

This approach was determined to be unfeasible for two reasons. First, there were situations where a single data stream in a decomposition could be represented by two equivalent link statements that the machine could not determine were equivalent. Figures 7 through 10 show a decomposition which illustrates this problem. The circled pairs of link statements in Figures 7 and 8 represent the same connections, but due to their different names the



input1.external-->Proc_a

input2.external-->Proc_b

ds_a.Proc_a:2s-->Proc_b

ds_b.Proc_a:2s-->Proc_b

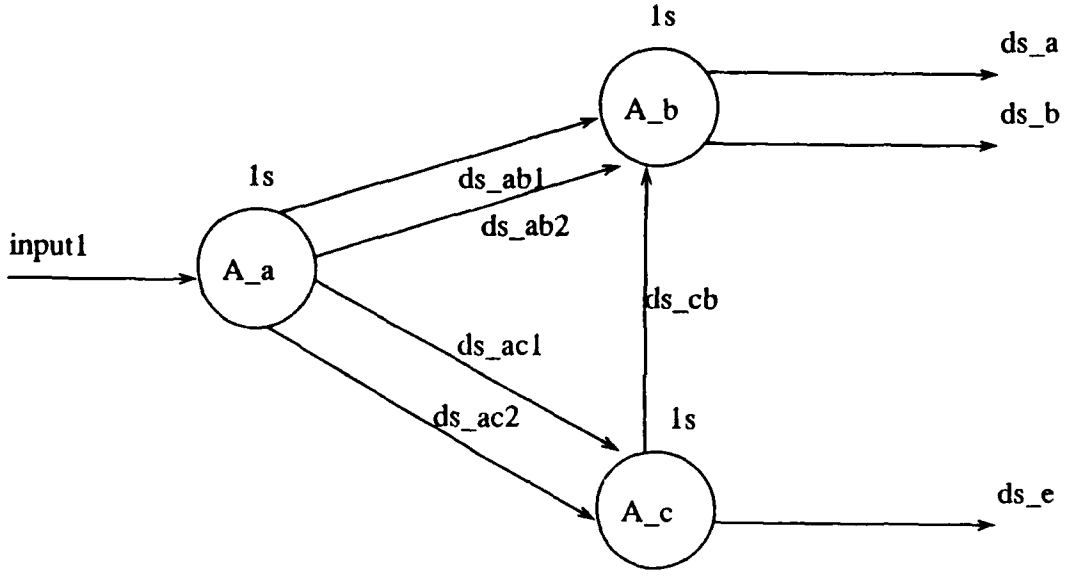
ds_e.Proc_a:2s-->Proc_c

ds_c.Proc_b:4s-->Proc_c

ds_d.Proc_b:4s-->Proc_c

output.Proc_c:2s-->external

Figure 7 Overall System Decomposed into its Component Operators



input1.external-->A_a

ds_ab1.A_a:1s-->A_b

ds_ab2.A_a:1s-->A_b

ds_ac1.A_a:1s-->A_c

ds_ac2.A_a:1s-->A_c

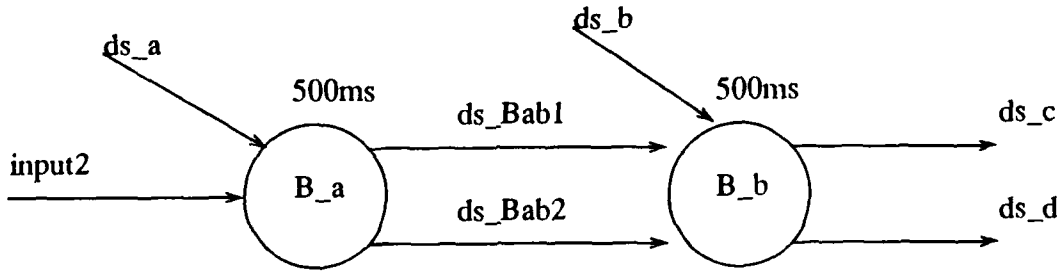
ds_cb.A_c:1s-->A_b

ds_a.A_b:1s-->Proc_b

ds_b.A_b:1s-->Proc_b

ds_e.A_c:1s-->Proc_c

Figure 8 Decomposition of Proc_a



input2.external-->B_a

ds_a.Proc_a:2s-->B_a

ds_b.Proc_a:2s-->B_b

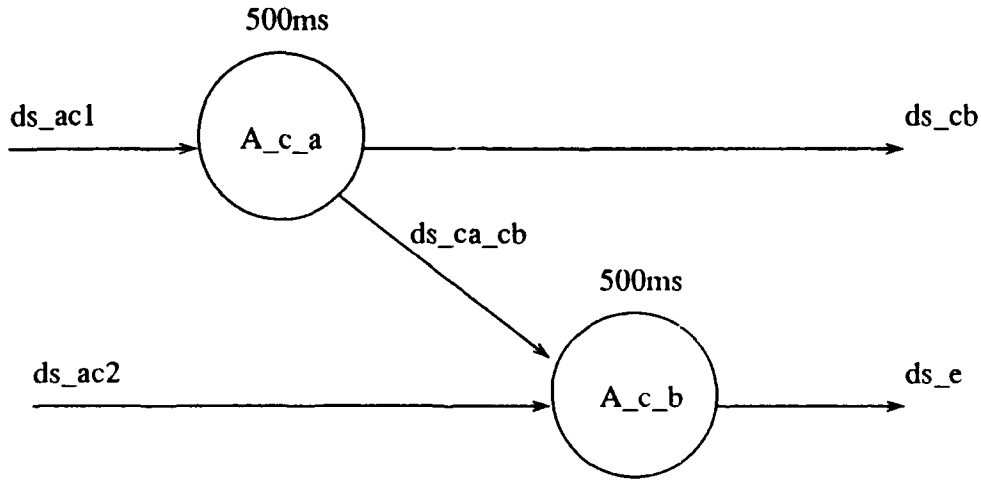
ds_Bab1.B_a:500ms-->B_b

ds_Bab2.B_a:500ms-->B_b

ds_c.B_b:500ms-->Proc_c

ds_d.B_b:500ms-->Proc_c

Figure 9 Decomposition of Proc_b



ds_ac1.A_a:1s-->A_c_a

ds_ac2.A_a:1s-->A_c_b

ds_ca_cb.A_c_a:500ms-->A_c_b

ds_cb.A_c_a:500ms-->A_b

ds_e.A_c_b:500ms-->Proc_c

Figure 10 Decomposition of A_c

machine could not possibly discover this. Second, sorting through all of the link statements associated with all of the atomic operators creates a potentially unsolvable problem.

A tree data structure can be used to assist in the creation of a schedule of operators. Store the name of the highest level operator in the root of the tree. Then determine the schedule for the operators in the root operator's decomposition. Store the names of these operators as children of the root operator in the order that they must be scheduled. Repeat this process for all operators until a tree containing all of the system's operators has been created. When the tree has been completely built, its frontier will contain a linear schedule of the operators (See Figure 11). Since this method does not require that one subtree have any knowledge of connections to other subtrees, there is no need for explicit sources and destinations for external inputs and outputs. The linked storage structure which is assembled by the graphical editor during a given operator's decomposition need only provide the information for showing relationships between the components of that operator's decomposition.

Based on these discoveries, the graphical editor was designed so that inputs and outputs for any decomposition come from and go to operators named EXTERNAL.

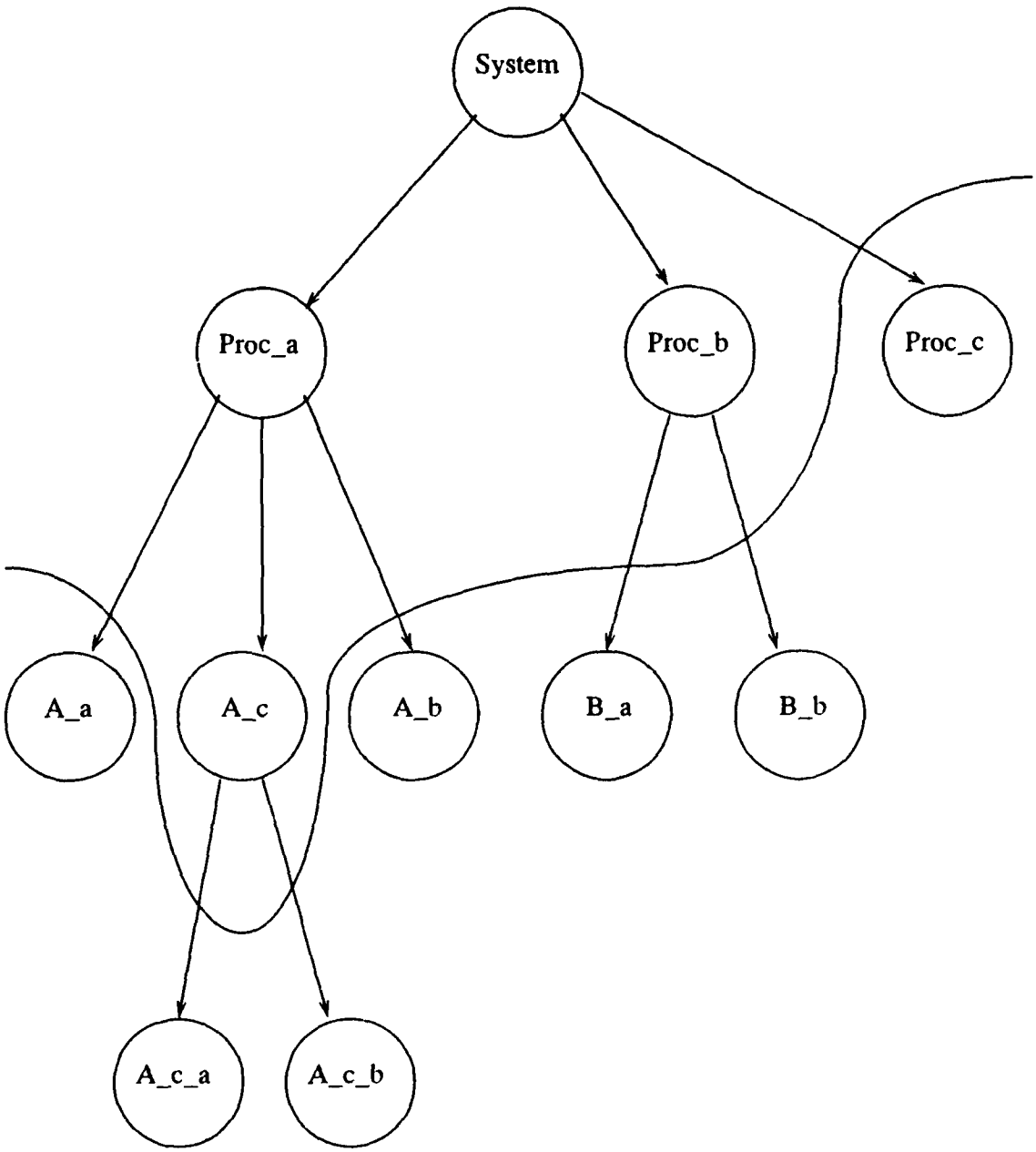


Figure 11 Storage of Operator Names in Tree Structure

B. INTERFACE DESIGN INFLUENCES

Four factors influenced the design of the graphical editor's user interface:

1) the choice of machines on which to implement CAPS
2) the choice of interface software support
3) human factors issues
4) user interface design guidelines

1. Sun Workstation

The Sun Workstation has many features which make it the machine of choice for the development and implementation of CAPS. The availability of a dedicated CPU in a multitasking environment greatly enhances the design team's ability to code, test and debug their software. The Sun Workstation provides a powerful integrated programming environment based on the unix operating system with its standard utilities which include:

1) Ada, C, Pascal and Fortran 77 compilers
2) vi and emacs editors
3) dbx debugger
4) make
5) revision control systems

and numerous others [Ref. 13].

2. Interface Software Support

a. Sun View

Sun View is a window based user environment which supports interactive graphics-based applications. There are

two aspects to Sun View. First, it is a user interface which provides multiple overlapping windows, each of which can run a task independent of the other windows. Second, it is a general toolkit for building window-based applications. A set of subroutine libraries provide three levels of abstraction for development of program interfaces [Ref. 14].

The highest level of abstraction for interface design is provided by the SUNTOOL library. It provides the functions of the User Interface Management System (UIMS) and also the functions necessary for the interactive creation of window applications. [Ref. 15]

The SUNWINDOWS library provides the middle level of interface abstraction. These functions are used to manage the hierarchy of overlapping windows. [Ref. 15]

The lowest level of abstraction is provided by the PIXRECT library. This library provides the functions needed for low level manipulation of pixels. [Ref. 15]

b. Interface Utilities

Sun View provides built in user interface utilities. The NOTIFIER is used to capture and distribute events among the multiple applications which are running. A SELECTION SERVICE is used to manage text selections. [Ref. 14]

c. Interface Building Blocks

Four types of windows are provided as building blocks for constructing user interfaces [Ref. 14]:

1) canvases provide an area on which programs can draw

2) text subwindows are used to display textual data

3) panels contain buttons and choice items

4) tty subwindows provide a capability to run other programs

d. The Sun View Model

Sun View is an object-oriented system whose objects are assembled in such a manner as to build a user interface. There are six classes of objects, the most important of which is a window (See Figure 12). Within the window class are the objects subwindow and frame. The subwindow objects are Canvas, Panel, Text and TTY.

Frames are window objects which themselves can be overlapped. They serve as a frame which surrounds a group of nonoverlapping subwindows, allowing them to be operated on as a unit.

Frames may also contain other frames. So the overall structure of Sun View is a tree of windows. The root of the tree is the base frame. The non-leaf nodes are frames and the leaf nodes are subwindows [Ref. 14]. Figure 13 shows the tree structure of the objects which comprise the graphical editor.

3. Human Factors

Since the graphical editor is a tool designed to aid a human in development of a software system prototype, human factors must be considered in its design.

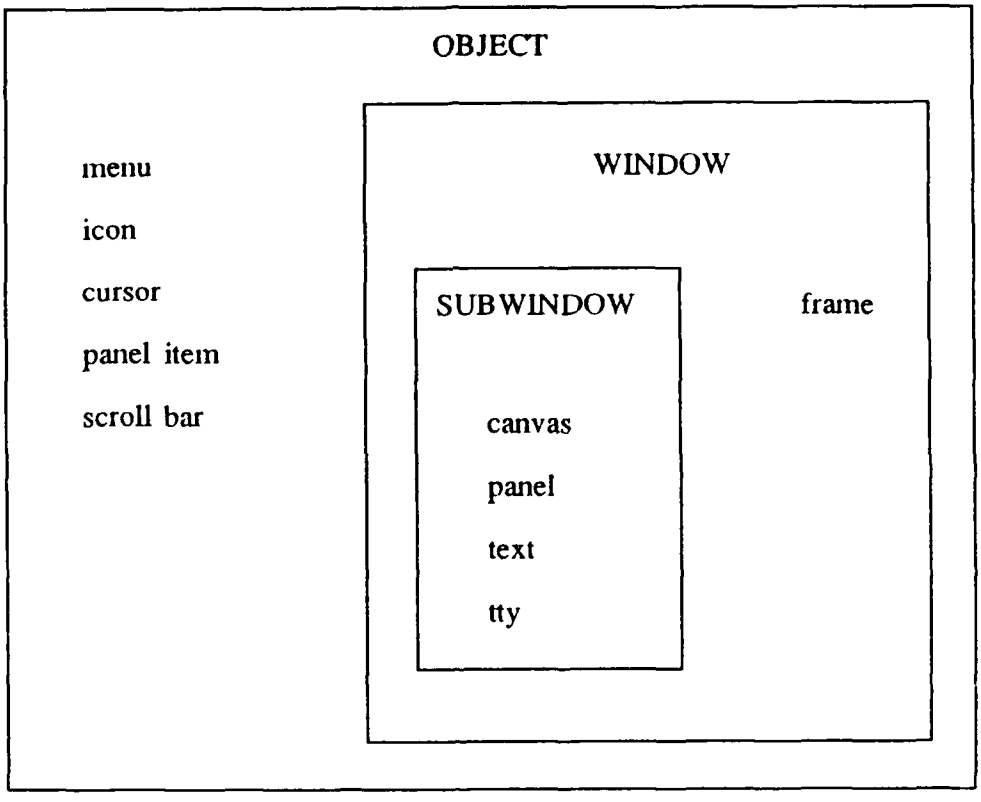


Figure 12 Sunview Objects [Ref. 14]

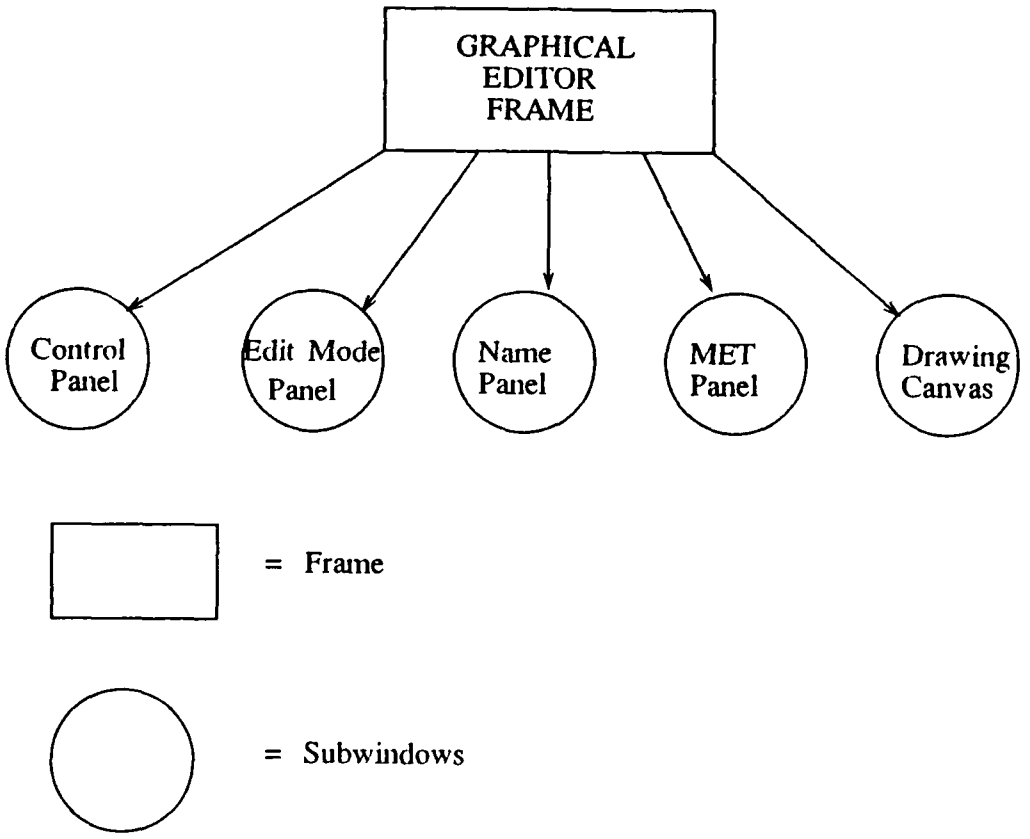


Figure 13 Tree Structure of Objects Comprising the Graphical Editor

a. Functional Principle

Systems with many controls can be very difficult to learn and use. To reduce this difficulty and also reduce the likelihood of errors, controls should be grouped according to their functionality [Ref. 16]. The graphical editor's controls fall into three functional groups (See Figure 14):

1) system control group
2) editing mode group
3) text input group

b. Sequence of Use Principle

Another way to improve the usability of a group of controls is to organize them in the same sequence that they are used. This improves the usability of a system by eliminating the need to jump around and therefore minimizing the amount of information the user must remember [Ref. 16]. The controls of the graphical editor are organized to be used in a series of top to bottom sequences. The top panel is used to control overall system functions: LOADING, STORING AND QUITTING. This panel is only used at the beginning and end of an editing session. The next panel down is the editing mode panel. It is used to switch from drawing one type of object to another. After the editing mode is changed, the user must enter a name and in the case of an operator, a time constraint. The input panels for these are therefore located immediately below the editing mode panel.

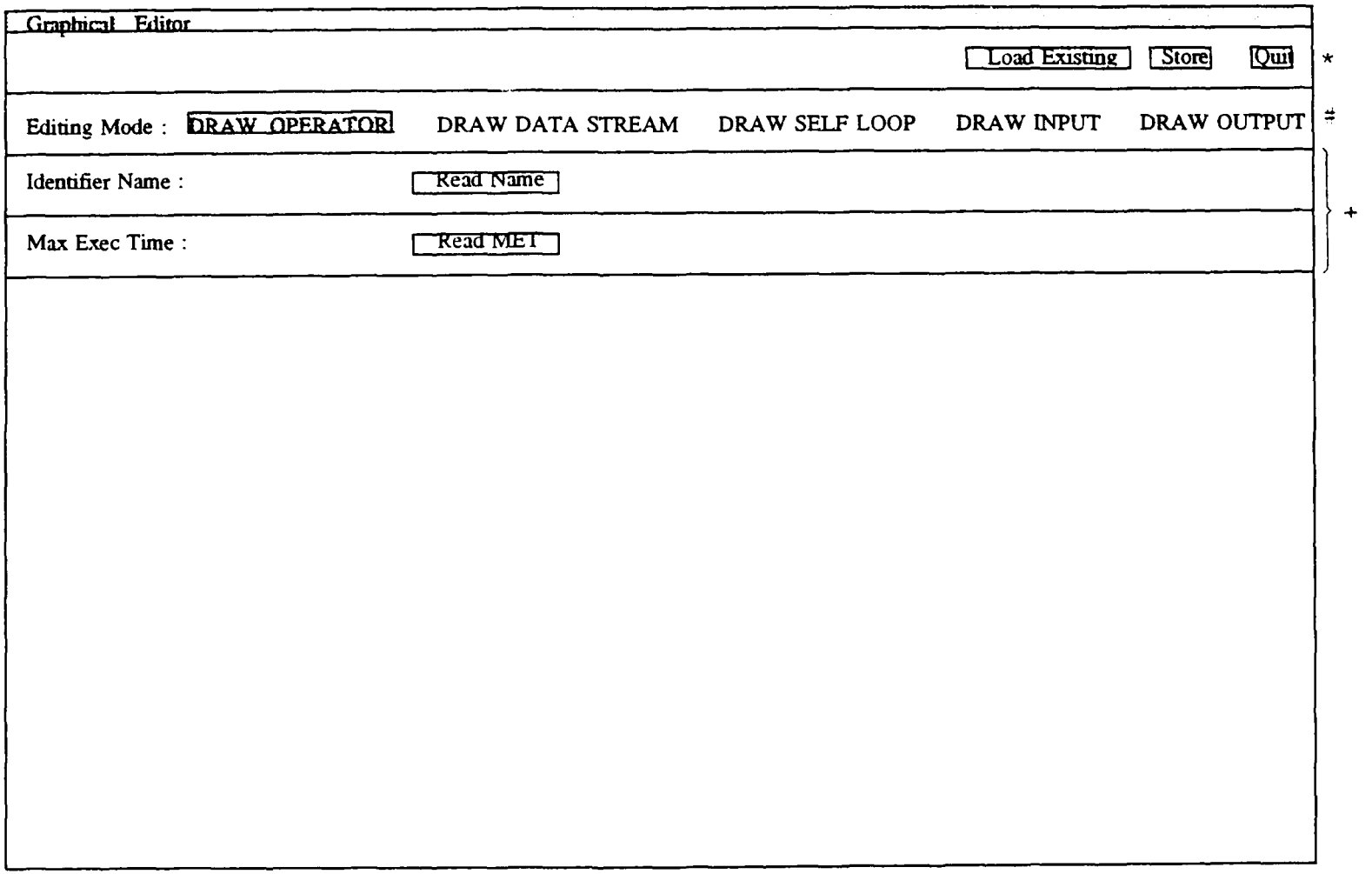


Figure 14 Functional Grouping of Controls

The drawing canvas is located immediately below the input panels. This ordering of controls always allows, but does not force, the user to operate in a top to bottom circular fashion as follows:

1) select mode (optional)
2) enter and read name
3) enter and read time constraint (if in operator mode)
4) draw object
5) repeat until done

c. Human Memory Capacity

Studies have shown that humans have the capacity to remember 7 ± 2 things [Ref. 16]. With this in mind, interface designers should keep the length of selection lists within this boundary if possible. The graphical editor was designed to meet this criteria. It consists of five basic parts and its longest menu has only five choices.

4. User Interface Design Guidelines

Reference 15 provides a list of guidelines which should be applied when designing a user interface.

1) Be intuitive (things should work as you would expect)
2) Accommodate experts and novices (provide confirmation override mechanisms)
3) Allow customization
4) Provide extensibility
5) Use lots of feedback (show status; make error messages clear)
6) Be predictable (use a consistent, easy to remember set of basic actions in obvious ways)

7) Be deterministic (consider type ahead and mouse ahead effects)

8) Avoid modes (if states that persist are necessary, make the feedback and exit path obvious)

9) Do not preempt the user (don't force them to respond)

C. INPUTS AND OUTPUTS

1. Inputs

The graphical editor accepts inputs from either the mouse or the keyboard. No restrictions are placed on the order that any of these inputs must occur except that a name must have been read before any type of object can be drawn. In the case of an operator a MET must also have been read. Figure 15 shows a black box representation of the graphical editor with all of its inputs and outputs.

An operating mode select event occurs when the mouse pointer is placed over one of the operating mode buttons and the left mouse button is pushed. This type of input selects whether the editor should (1) load an existing diagram, (2) store the current diagram or (3) quit. The default mode is for the editor to be ready to create a new diagram.

An editing mode select event is also a mouse input. This input establishes the context in which canvas events will be interpreted.

To draw an object the user positions the mouse pointer in the canvas area, pushes the left mouse button down, moves it to another location and releases it. Releasing the left mouse button causes an end of object event

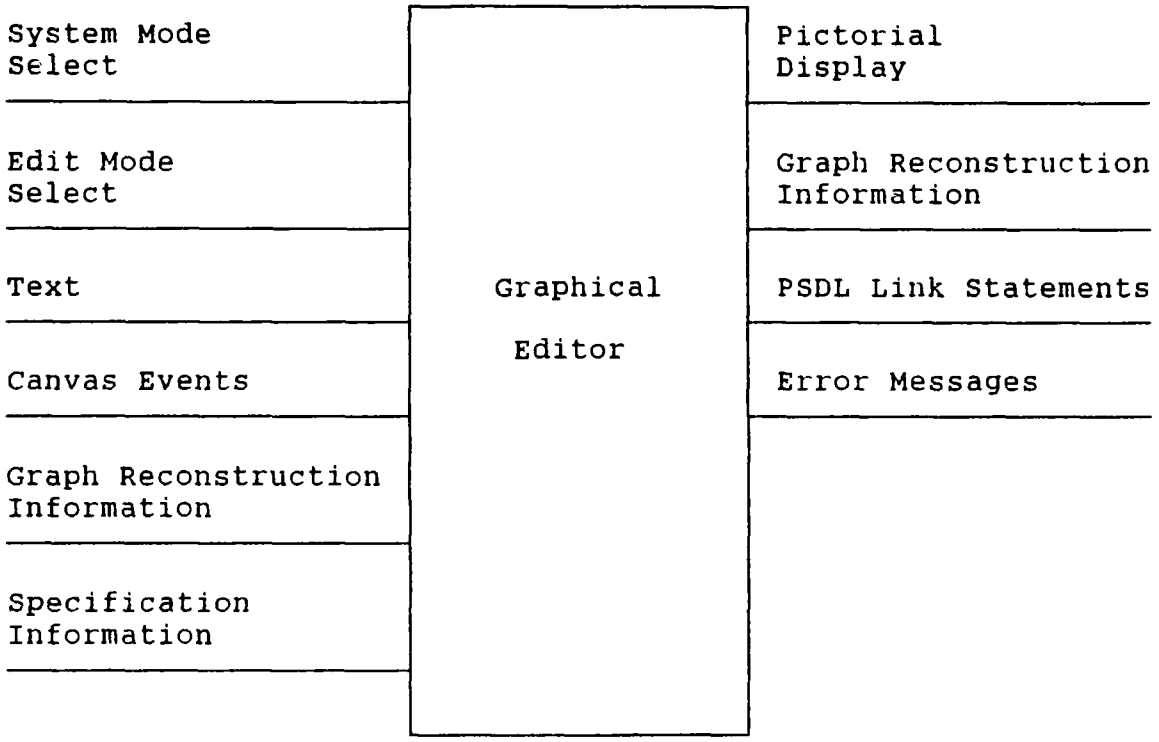


Figure 15 Inputs and Outputs of the Graphical Editor

which tells the editor to draw the object using the start and stop locations.

To delete an object the user positions the mouse on the object to be deleted and initiates a right mouse button down event. This input will erase the object from the display and free up its storage.

Textual inputs to the graphical editor are typed in via the keyboard. These inputs will only be accepted if the mouse pointer is located in one of the text panels. After text input has been entered, it must be validated by causing a read event. This is done by positioning the mouse pointer over the appropriate read button and pushing the left mouse down.

If the graphical editor is going to be used to edit an existing diagram, the user interface function must retrieve the necessary reconstruction information from the design database and store the information in a file named graph.pic prior to invoking the editor [Ref. 6]. The graphical editor then reads in this information and reconstructs the diagram.

2. Outputs

The graphical editor has two kinds of outputs: visual and textual. If the user draws an object, it is displayed on the canvas so that he can see it. If a user generated error occurs, an error message will immediately be

displayed at the top of the canvas. Once the error condition has been corrected, the error message disappears.

When the mouse is in a particular subwindow of the display, visual feedback in the form of a bold subwindow border, is provided.

After a decomposition has been completed and the user has selected store, the editor will generate two kinds of textual output. The PSDL link statements will be written to the file graph.link. Also, the information needed to reconstruct the diagram will be written to the file graph.pic. The user interface will store this information in the design database [Ref. 6].

D. DATA STRUCTURES

The primary storage structure for the graphical editor is a linked list of operators. A single structure named Operator_list has pointers to the head and tail elements of this list. Each element in the operator list is a structure of type Operator. The Operator structure has ten fields (See Figure 16).

The name field is a pointer to a name structure which is comprised of an 40 character array and a field specifying the length of the name. The optype field is used to signify whether the structure represents an actual operator or whether it represents a NULL operator which is used as the source for external inputs. The xstart and ystart fields specify the starting (x,y) coordinates of the operator. The

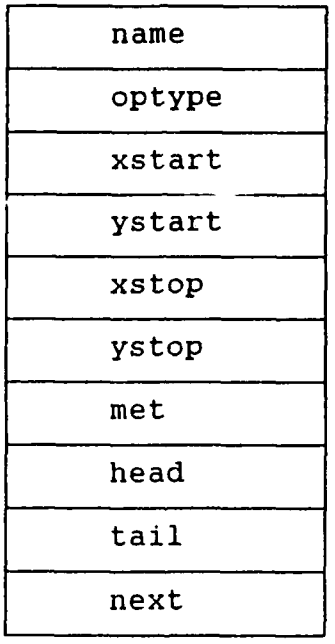


Figure 16 Operator Storage Structure

xstop and ystop fields contain the operator's stopping (x,y) coordinates. The met field contains a pointer to a Met structure which is identical to a Name structure. The last three fields are pointers. The head and tail fields point to a linked list of lines which leave this operator. The next field points to the next operator in the linked list of operators.

As mentioned in the previous paragraph, each operator has an associated line list. This line list is a linked list of structures of type Line (See Figure 17). These lines originate at the operator to which they are attached.

The name, xstart, ystart, xstop, ystop and next fields of the Line structure serve the same purpose as they did in the Operator structure. The lntype field is used to identify whether the line is an input, output, data stream or a state. The dest field is used to point to a name structure which holds the name of the operator on which the line terminates. For output lines, this field will always point to a name structure containing the name EXTERNAL. Figure 18 shows a decomposition of an operator. Figure 19 shows the resulting storage structure generated during the decomposition process.

E. ALGORITHM FOR THE GRAPHICAL EDITOR

At a very high level, the algorithm for the graphical editor is simply:

1) create the window
2) poll for events

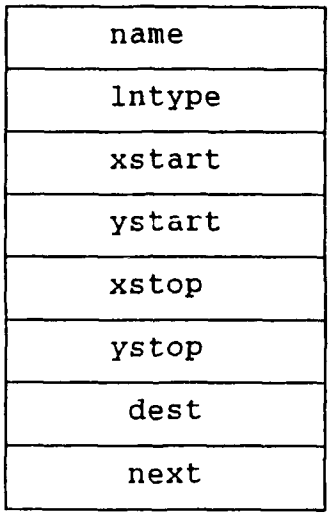


Figure 17 Line Storage Structure

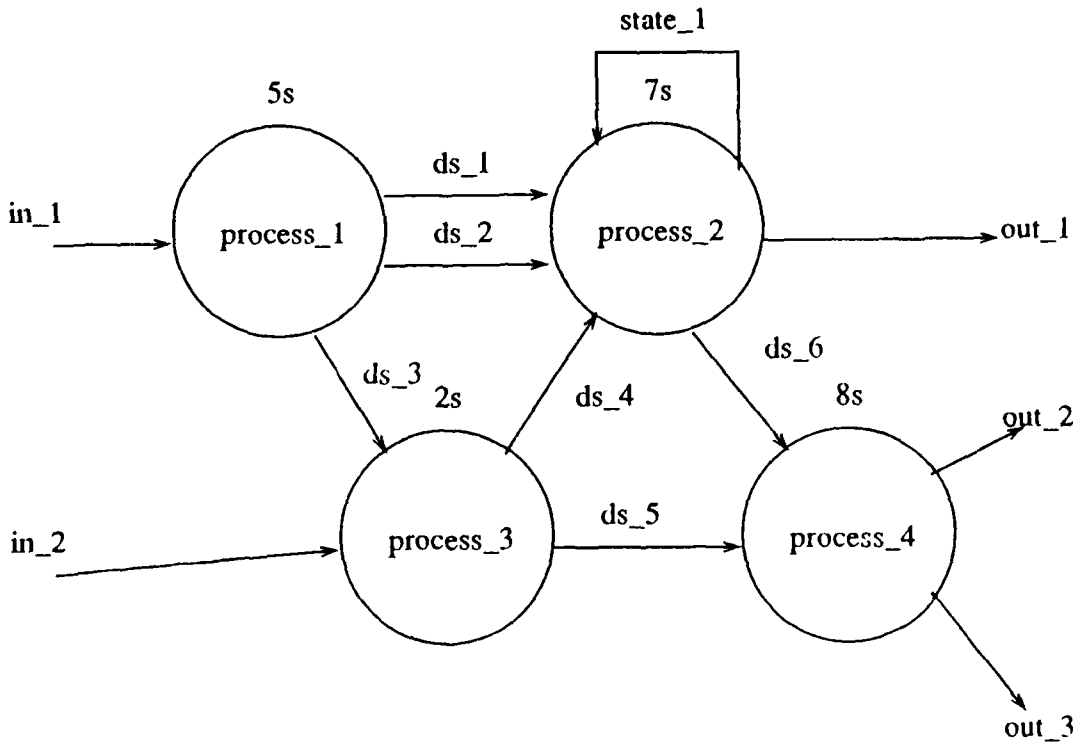


Figure 18 Decomposition of an Operator

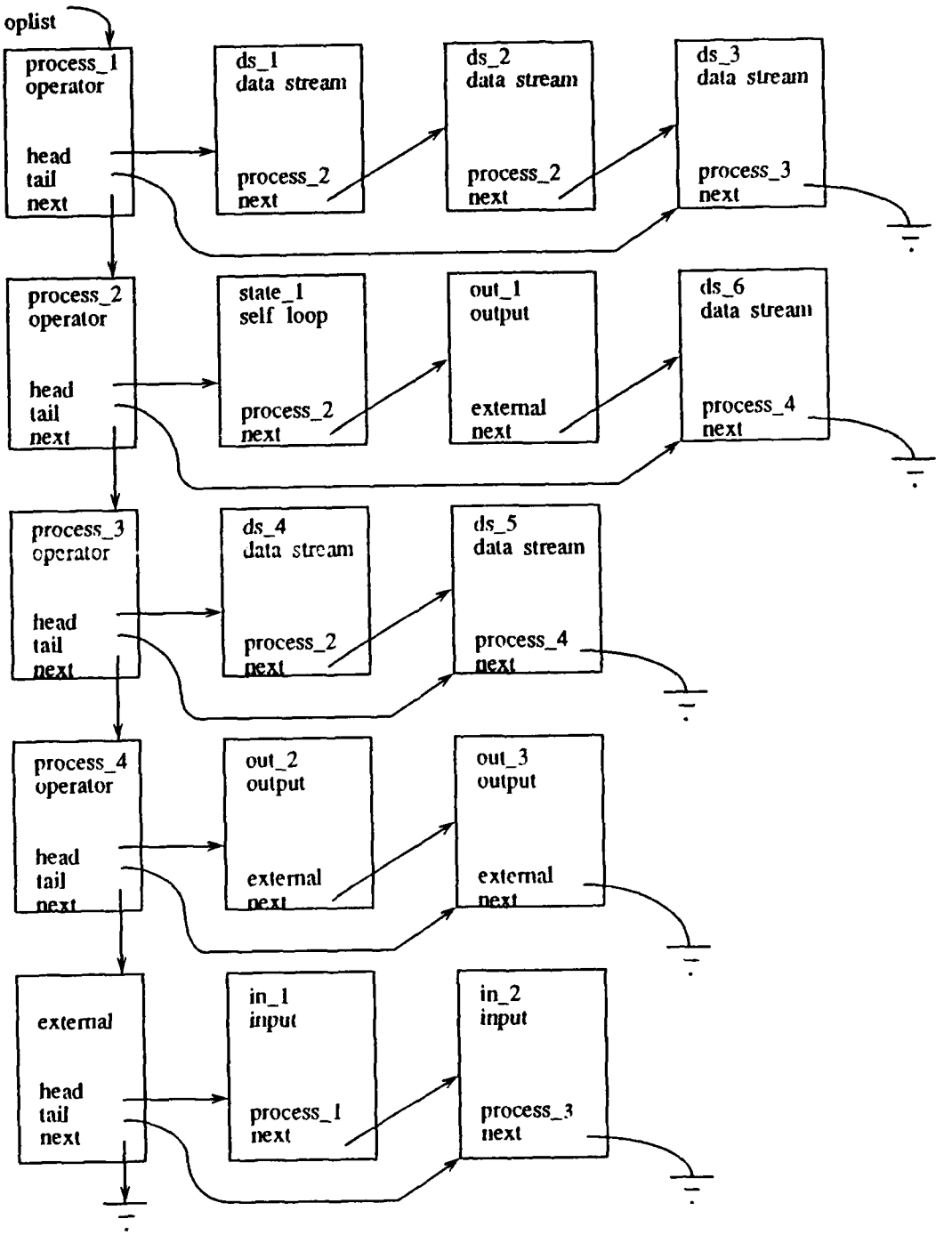


Figure 19 Storage Structure Resulting from Operator Decomposition

Sun View has built in routines which allow the interface designer to construct an interactive window application. The interface designer need only provide the routines for handling the details of the application. The Appendix contains a program listing of the graphical editor.

1. **Create the User Interface**

The user interface for the graphical editor is a window comprised of five subwindows, each of which is one of the Sun View application building blocks.

The first step in creating a Sun View application is to create a frame. This is done with the Sun Window routine window_create. Parameters for this routine allow the specification of various attributes for the frame object. These attributes include its name, icon, size, location, window type, location on the screen and numerous others.

After the frame has been created, the routine window_create is used to create each of the subwindows which are used to fill in the frame. The graphical editor frame is tiled in with four panel subwindows and a drawing canvas subwindow.

2. **Poll for Events**

In conventional interactive programming the main event polling loop resides within the application program. A notification-based system such as Sun View has the polling loop in a notifier. The notifier reads events and then notifies the appropriate application procedure that input has

occurred. This scheme requires that each procedure must be registered with the notifier so that it knows who to call for a particular event. The advantage to this is that the system takes over the job of managing the event-driven environment. [Ref. 14]

The Sun Window routine **panel_create_item** is used to build a panel subwindow. The parameters for this procedure include its name, image type, image label and the procedure to be notified when the button is pushed. So procedures which are to be called as a result of a button being pushed are registered with the notifier by the panel_create_item routine.

a. System Control Events for the Graphical Editor

The user of the graphical editor must have a means of retrieving previously generated diagrams and storing new or altered diagrams. He must also have some way to terminate the editor. These capabilities are provided by the following buttons which are located in the top subwindow of the graphical editor frame:

1) load existing
2) store
3) quit

When the graphical editor is started it comes up in a mode which allows the user to create a new decomposition diagram.

(1) Load Existing. The graphical editor can also be started for the purpose of editing a previously drawn

decomposition. To load the diagram the user must click on the **load_existing** button. This will cause the routine load_proc to read the file graph.pic which contains the reconstruction data. This data must have been previously retrieved from the design database by the user interface [Ref. 6]. When the load_proc routine reads in the information for the graphical object, it checks to see if the object is an operator. If it is an operator, an operator storage element is created, filled in with its information, and is attached to the list of operators. If the object is of type **EXTERNAL**, an operator element is also created and is linked to the operator list. EXTERNALs are NULL operator nodes which serve as the source operator for input lines. The only fields of an EXTERNAL which get useful values are those pointing at the line list.

Any object encountered during the load process which is not an operator or external is some type of line. Therefore, a line storage element is created, its values are filled in and it is linked to the line list of the last operator which was read in.

After all of the objects in the file being loaded have been read in, stored and linked, the diagram is ready to be drawn. Load_proc's final action is to call the routine **redraw_diagram** which traverses the entire linked storage structure and draws each object.

(2) Store. After the user has completed drawing or altering a decomposition diagram, two things must be done. First, the information in the diagram must be stored in a manner which allows it to be redrawn. Second, the diagram must be converted into the equivalent PSDL link statements. These link statements will be attached to the implementation part of the PSDL specification file by the sequence control function [Ref. 6].

Information for diagram reconstruction is stored by the routine **store_diagram**. This routine does a traversal of the storage structure, writing out the contents of each operator node immediately followed by the contents of each of its associated line nodes.

Creating the PSDL link statements is a similar process. The **create_PSDL** routine traverses the entire storage structure, generating a PSDL link statement for each line node it finds. The link statement is a string of characters which result from the concatenation of the following six substrings:

1) the line name (stored in the line node)
2) the character "."
3) the source operator's name (the name of the operator whose lines are being processed)
4) an optional ":" and MET (if the source operator has an MET)
5) the character string"-->"
6) the destination operator's name (stored in one of the line nodes fields)

After the diagram has been stored, the **store_diagram** routine tells the system that it is safe to exit.

(3) Quit. The user may terminate the editor by clicking on the **quit** button. This action will cause the routine **quit_proc** to execute. This routine will first check to see if the diagram has been stored. If so, it will destroy the window. If the diagram has not been saved, an error message will appear on the drawing canvas telling the user to store the diagram. The user can terminate the session without saving by positioning his mouse pointer on the black bar at the top of the window and pushing his right mouse. A pop-up menu will appear which has **quit** as one of the choices. Selecting this will circumvent the storage check and kill the editor.

b. Mode Select Events

The graphical editor always starts in the **DRAW OPERATOR** mode. This is because operators must be drawn before data streams. This requirement has the advantage of making it easy to check the syntax of the diagram. The editor ensures that lines intersect operators in the way which is appropriate to their type (i.e. input, output, etc.).

To switch operating modes, the user clicks on the desired mode. The selector will switch to reverse video indicating that it has been selected. The selection causes

the notifier to call the **mode_select** routine which sets the global **edit_mode** variable to the appropriate value. This establishes the context in which canvas events for the left mouse button will be interpreted.

c. **Text Panel Events**

The graphical editor has two panels which provide a means of entering textual information.

(1) Name Panel. The name panel allows the user to enter a name for an operator or a line. After the name has been typed in, the **read_name** button must be selected. This action causes the notifier to inform the routine **input_name** to read the panel and also causes the routine **is_valid_ada_id** to check the syntax of the name. If the name is not a valid Ada identifier, an error message is displayed and the system rejects drawing events on the canvas.

(2) MET Panel. The MET panel works essentially the same as the name panel. The only difference is that the routine **is_valid_MET** is used to check that the value is an integer and that it has the proper units. Invalid values result in an error message and a lockout of operator events from the canvas since only operators have an MET.

d. **Canvas Events**

The graphical editor screens the event handler for left mouse down events, left mouse drag events, left mouse up events and right mouse down events.

(1) Left Mouse Down. If the process_canvas_events routine receives a left mouse down event it will capture the x and y coordinates of the position where the event occurred. These values are stored and become the starting position of the object being drawn. Which object is drawn depends upon the current editing mode.

(2) Left Mouse Drag Events. If the left mouse is down and drag events are detected, the process_canvas_events routine will rubberband the object. As the mouse pointer is moved across the screen process_canvas_events repeatedly captures the position of the pointer. For each new position, the routine rubberband is called to blank out the previous version of the object and then redraw it using the most recent starting and stopping coordinates. The result is that the line is erased and redrawn as fast as the user moves the pointer across the screen.

(3) Left Mouse Up Events. When a left mouse up event occurs, the process_canvas_events routine calls rubberband a last time to delete the last rubberbanded version then captures the final stopping coordinate. The routine process_object then performs the syntactic and semantic checks on the object.

(a) Processing Operators. If the editing mode is DRAW OPERATOR, process_object will verify that a name and a MET are available and that the coordinates of the new

operator do not overlap another operator. When all of these conditions are satisfactory it calls the routine **process_operator**. This routine will cause the following seven steps to take place:

1) the object will be drawn
2) the MET will be retrieved
3) the name will be retrieved
4) the name will be displayed, centered in the operator
5) the MET will be displayed, centered over the operator
6) the operator will be allocated storage and stored
7) the stored operator will be appended to the list of operators

(b) Processing Lines. If the editing mode is DRAW DATA STREAM, DRAW INPUT, DRAW OUTPUT or DRAW SELF LOOP, routine process_object will verify availability of a legal Ada identifier and will ensure the line intersects an operator in the appropriate fashion. If the checks turn out satisfactory, the routine **process_line** is called. This routine will cause the following actions:

1) draw the appropriate line
2) draw the arrowhead on the end of the line
3) if the line is an input line:
   a. creates a NULL operator to act as its source
   b. links the NULL operator to the operator list
4) retrieves the line's name
5) displays the name on the line
6) creates the storage for the line and fills in the values

7) appends the line to the source operator's list of lines

(4) Right Mouse Down Events. When a right mouse down event occurs, the routine process_canvas_events checks to see if the mouse's coordinates are within the pick criteria of either a line or an operator. If so, the object is deleted from the storage structure by routine **delete_line** or **delete_op** as appropriate. After the deletion is complete, routine redraw_diagram draws the remaining objects.

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis shows that a graphical editor for performing hierarchical decomposition of composite PSDL operators can be designed and implemented. It also demonstrates that a linked list of operators each containing a linked list of outputs is an effective method of storing the graphical elements. This is evidenced by the speed which diagrams can be loaded, stored, redrawn and deleted from. This method of storage is also shown to support the transformation of a pictorial diagram into textual PSDL link statements.

It also shows that through the use of modes, the graphical editor can perform syntactic and semantic checks during the decomposition. These checks ensure that the PSDL link statements are always of the proper form and convey the designer's intended meaning.

Research on the graphical editor, as it relates to PSDL, indicates that the prototype design will take shape in the editor. Therefore, there is a need to start new specification files for each new operator introduced during a decomposition. Each of these new files will contain the operator's name, inputs, outputs, states and MET.

This research indicates that using the graphical editor to randomly edit up and down through the objects in the design database has the potential to cause inconsistencies

between the specification of an operator and its pictorial values. A better method would be to make corrections in a top-down fashion, alternating between graphical editor and the syntax directed editor for each object. This method, although slower, would ensure view consistency between the textual and graphical representations of the decomposition.

B. RECOMMENDATIONS

1) The requirement that forces the user to enter the textual values for an object before it is drawn decreases the user friendliness of the graphical editor. A method should be developed which allows entry of this textual information at any time during the decomposition process.

2) Work should be done to make the system configurable. A user should be able to disable checking and therefore enter any object in any order desired. This would greatly improve its appeal to the expert user.

3) The process of decomposing an operator is like programming with objects. A better way to enter an operator's specification might be to open a window within the graphical editor which allows the user to define the object as it is being created instead of using a syntax directed editor to enter its specification.

4) Currently the graphical editor only supports prototypes written in Ada. Providing a panel with a choice of languages, which would enable an appropriate identifier syntax checker, would make the editor less language specific.

5) Currently all error messages are displayed on the drawing canvas. A better method would be to display them in dialogue boxes.

6) The user should be able to just "pick" the names of all external data streams from a list of inputs and outputs, that are passed to the editor when it is invoked, rather than being required to type them. This would eliminate another possible source of errors.

LIST OF REFERENCES

1. Luqi, Ketabchi, M., "A Computer Aided Prototyping System," IEEE Software, v. 5, pp. 66-72, March 1988.

2. Whitten, J. L., Gentley, L. D., and Ho, T. I. M., Systems Analysis and Design Methods, Times Mirror/Mosby College Publishing, 1986.

3. Luqi, Rapid Prototyping for Large Software System Design, Ph. D. Dissertation, University of Minnesota, 1986.

4. Booch, G., Software Engineering with Ada, Benjamin/Cummings Publishing Co., 1987.

5. Luqi, Berzins, V., Yeh, R., "A Prototyping Language for Real-Time Software," IEEE Transactions on Software Engineering, pp. 1409-1423, October 1988.

6. Raum, H., The Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

7. Janson, D. M., A Static Scheduler for the Computer Aided Prototyping System: An Implementation Guide, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

8. Moffitt, C. R., A Language Translator for a Computer Aided Rapid Prototyping system, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

9. Altizer, C. E., Implementation of a Language Translator for a Computer Aided Rapid Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

10. Marlowe, L. C., A Scheduler for Critical Timing Constraints, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

11. Department of Defense Military Standard/American National Standards Institute Standard ANSI/MIL-STD-1815A-1983, Reference Manual for the Ada Programming Language, 17 February 1983.

12. Reiss, S. P., "GARDEN Tools: Support for Graphical Programming," Advanced Programming Environments. Proceedings Vol 244, Edited by Conradi, R., Didriksen, T. M., and Wanvik, D. H., Springer-Verlag, 1986.

13. Adams, E., and others, "SunPro: Engineering a Practical Program Development Environment," Advanced Programming Environments. Proceedings Vol 244, Edited by Conradi, R., Didriksen, T. M., and Wanvik, D. H., Springer-Verlag, 1986.

14. Sun Microsystems, Inc., SunView Programmer's Guide, Revision: A, pp. 3-24, 15 October 1986.

15. Hopgood, F. R. A., and others, Methodology of Window Management, Springer-Verlag, 1986.

16. Sanders, M. S., and McCormick, E. J., Human Factors in Engineering and Design, 6th ed., McGraw-Hill Book Co., 1987.

APPENDIX

PROGRAM TEXT FOR THE GRAPHICAL EDITOR

```
#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/panel.h>
#include <suntool/seln.h>
#include <math.h>
#include <string.h>
#include <ctype.h>

                                /* define constants for the object types */
#define OPERATOR    0
#define DATA_STREAM 1
#define SELF_LOOP   2
#define INPUT       3
#define OUTPUT      4
#define EXTERNAL    5

#define PI 3.141592654                                  /* constant PI */

#define DISP_WIDTH  140                                /* display width */
#define DISP_HT     50                                /* display height */

#define ARROW_LENGTH 9                      /* length of the arrow head */
#define TEXT_MAX_LEN 35              /* length of name which is visible */
#define TIME_MAX_LEN 10           /* length of the met which is visible */

#define PROXIMITY  25                /* define size of line pick region */

                                     /* import predefined editor icon */
static short editor_icon[] = {
#include "/usr/suns2/thorsten/graphics/editor.icon"
};

DEFINE_ICON_FROM_IMAGE(editor, editor_icon);


Frame    frame;                                      /* handle for frame */

Panel    op_mode_panel,                       /* handle for top panel */
         edit_mode_panel,                    /* handle for side panel */
         met_panel,                           /* handle for met panel */
         name_panel;                         /* handle for name panel */

Canvas   drawing_canvas;                   /* handle for drawing canvas */

Event    *event                                      /* handle for events */
```

```
Pixfont  *bold;                                  /* handle for borders */

Pixwin   *drawing_pw;                       /* handle for drawing pixwin */

int      edit_mode;           /* global - stores the current edit mode */

int      name_checked = 0,      /* global - signals that name is valid */
         met_checked = 0;        /* global - signals that met is valid */

int      graph_saved = 0;             /* global - signals saved status */

Panel_item object_name,                             /* handle for name */
           time_constraint;                          /* handle for met */

char    *name_buf,            /* global - buffer to read the name into */
        *met_buf;              /* global - buffer to read the met into */

FILE    *f,                                  /* PSDL link statement file */
        *g;                /* file which holds the reconstruction info */

typedef struct                     /* storage struct for names and mets */
{
    int   length;                                       /* string length */
    char  string[40];          /* array to hold the name or met string */
}Name,Met;

typedef struct line                           /* line storage structure */
{
    Name  *name;                                         /* name of line */
    int   lntype;                 /* identifies the type of line it is */
    int   xstart;                    /* x coord of line starting posit */
    int   ystart;                    /* y coord of line starting posit */
    int   xstop;                     /* x coord of line stopping posit */
    int   ystop;                     /* y coord of line stopping posit */
    Name  *dest;                  /* operator on which line terminates */
    struct line  *next;           /* pointer to next line in this list */
}Line;


typedef struct operator                   /* operator storage stucture */
{
    Name  *name;                                    /* operator's name */
    int   optype;       /* identifies contents as operator or external */
    int   xstart;  /* x coord where operator should start to be drawn */
    int   ystart;  /* y coord where operator should start to be drawn */
    int   xstop;          /* x coord of the operator's opposite corner */
    int   ystop;          /* y coord of the operator's opposite corner */
    Met   *met;             /* maximum execution time for the operator */
    Line  *head;                 /* head of the operator's output list */
    Line  *tail;                 /* tail of the operator's output list */
```

```
    struct operator *next;        /* pointer to next operator in list */
}Operator;


typedef struct                              /* the list for operators */
{
    Operator *head;               /* pointer to the head of the list */
    Operator *tail;               /* pointer to the tail of the list */
}Operator_list;


Operator_list operator_list;
Operator_list *op_list = &operator_list;
Operator      *op, *sop, *dop;

Name          name_pointer;
Name          *name = &name_pointer;
Met           *tc;
Line          *ln;

                                  /* forward declarations of functions */

static    Notify_value process_canvas_events();
static    Notify_value mode_select();

Operator *alloc_operator();
Operator *pick_operator();
Operator *create_op();

Line     *alloc_line();
Line     *pick_line();
Line     *create_line();

Name     *external();
Name     *get_name();

Met      *get_met();

int       is_op_pick();
int       is_line_pick();
int       is_valid_ada_id();
int       is_valid_met();

                                 /* forward declarations of procedures */
help_proc();
quit_proc();
load_proc();
store_proc();
input_name();
input_met();
append_line_to_op();
```

```
is_op_pick();
is_input_pick();
process_operator();
process_line();
rubber_band();
redraw_diagram();
delete_line();
delete_op();
delete_input_lines();
display_error_msg();
display_name();
display_met();
draw_arrowhead();
draw_object();
create_PSDL();
store_diagram();
```

```
main(argc, argv)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: create_operating_mode_panel()
          create_editing_mode_panel()
          create_name_panel()
          create_met_panel()
          process_canvas_events()

   CALLED BY: N/A

   PURPOSE: creates the frame and the subwindows which comprise the
            graphical editor. It also registers the routine
            process_canvas_events with the notifier.

*************************************************************************/

int    argc;
char **argv;

{
                        /* cause borders to highlight if region entered */
    bold = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.b.12");
    if (bold == NULL) exit(1);

                                        /* create the outer display frame */
    frame = window_create(NULL,FRAME,
                FRAME_LABEL,              "graphical editor",
                FRAME_ICON,               &editor,
                FRAME_CMDLINE_HELP_PROC, help_proc,
                FRAME_ARGS,               argc, argv,
                WIN_ERROR_MSG,            "can't create window.",
                WIN_X,                    15,
                WIN_Y,                    15,
                WIN_ROWS,                 DISP_HT,
                WIN_COLUMNS,              DISP_WIDTH,
                0);

                                           /* create op_mode_panel */
    op_mode_panel = window_create(frame, PANEL, WIN_FONT , bold, 0);
    create_operating_mode_panel();

                                       /* create editing mode panel */
    edit_mode_panel = window_create(frame, PANEL, WIN_FONT, bold, 0);
    create_editing_mode_panel();

                                    /* create panel to read object names */
```

```
    name_panel = window_create(frame, PANEL, WIN_FONT, bold, 0);
    create_name_panel();

                                /* create panel to read operator's met */
    met_panel = window_create(frame, PANEL, WIN_FONT, bold, 0);
    create_met_panel();

                                            /* create canvas to draw on */
    drawing_canvas = window_create(frame, CANVAS, WIN_FONT, bold,
                            WIN_CONSUME_KBD_EVENT, WIN_ASCII_EVENTS,
                            WIN_EVENT_PROC, process_canvas_events,
                            CANVAS_RETAINED, TRUE, 0);

                                    /* cause drag events to be accepted */
    window_set(drawing_canvas,WIN_CONSUME_PICK_EVENT, LOC_DRAG, 0);

                                                /* define drawing pixwin */
    drawing_pw = canvas_pixwin(drawing_canvas);

                                        /* initialize the operator list */
    operator_list.head = operator_list.tail = NULL;

                                        /* poll for events in the frame */
    window_main_loop(frame);

}  /* end main */
```

```
create_operating_mode_panel()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: load_proc()
          store_proc()
          quit_proc()

   CALLED BY: main()

   PURPOSE: This routine builds the top panel of the display which
            contains the operating mode buttons. It also registers
            the routines load_ proc, store_proc, and quit_proc with
            the notifier.

*************************************************************************/
{
                                                /* display panel message */
    panel_create_item(op_mode_panel, PANEL_MESSAGE,
        PANEL_LABEL_STRING,
      "
        (left mouse selects)   ", 0);

                          /* create button to load reconstruction info */
    panel_create_item(op_mode_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(op_mode_panel,
        "Load Existing", 0,        0),
        PANEL_NOTIFY_PROC, load_proc, 0);

                                  /* create button to store the diagram */
    panel_create_item(op_mode_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(op_mode_panel,
        "Store",    0, 0), PANEL_NOTIFY_PROC, store_proc, 0);

                              /* create button to terminate the program */
    panel_create_item(op_mode_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(op_mode_panel,
        "Quit", 0, 0),
        PANEL_NOTIFY_PROC, quit_proc, 0);

                                    /* fit border around the top panel */
    window_fit_height(op_mode_panel);

}  /* end create_operating_mode_panel */
```

```
create_editing_mode_panel()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: mode_select()

   CALLED BY: main()

   PURPOSE: This routine builds the edit mode panel and registers the
            routine mode_select with the notifier.

*************************************************************************/
{
                                          /* put blank line in panel */
    panel_create_item(edit_mode_panel, PANEL_MESSAGE,
        PANEL_LABEL_STRING,
        "
                                                              ",
        0);

                                        /* create the mode select panel */
    panel_create_item(edit_mode_panel, PANEL_CHOICE,
            PANEL_LABEL_STRING,      "   Editing Mode:  ",
            PANEL_CHOICE_STRINGS,    "    Draw Operator     ",
                                     "  Draw Data Stream    ",
                                     "   Draw Self Loop     ",
                                     "      Draw Input      ",
                                     "      Draw Output     ",
                                     0,
            PANEL_FEEDBACK,          PANEL_INVERTED,
            PANEL_NOTIFY_PROC,       mode_select, 0);

                                   /* insert a blank line below the menu */
    panel_create_item(edit_mode_panel, PANEL_MESSAGE,
        PANEL_LABEL_STRING,
        "
                                                                  ",
        0);

                              /* fit window around the editing mode panel */
    window_fit_height(edit_mode_panel);

}  /* end create_editing_mode_panel */
```

```
create_name_panel()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: input_name()

   CALLED BY: main()

   PURPOSE: This routine creates the panel which reads the names which
            are assigned to objects. It also registers the routine
            input_name with the notifier.

*************************************************************************/
{
                                                    /* put in a header */
    object_name = panel_create_item(name_panel, PANEL_TEXT,
        PANEL_LABEL_STRING,         " Identifier Name:",
        PANEL_VALUE,                "",
        PANEL_VALUE_DISPLAY_LENGTH,TEXT_MAX_LEN,
        0);

      /* create read button and register input text with the notifier */
    panel_create_item(name_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(name_panel,
        "read name", 0,0),
        PANEL_NOTIFY_PROC, input_name,
        0);

                           /* put top and bottom boundary on the panel */
    window_fit_height(name_panel);

}  /* end create_name_panel */
```

```
create_met_panel()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: input_met()

   CALLED BY: main()

   PURPOSE: This routine creates the panel which reads the maximum
            execution times which are assigned to the operators. It
            also registers the routine input_met with the notifier.

*************************************************************************/
{
                                                    /* put in a header */
    time_constraint = panel_create_item(met_panel, PANEL_TEXT,
        PANEL_LABEL_STRING,         " Max Exec Time  :",
        PANEL_VALUE,                "",
        PANEL_VALUE_DISPLAY_LENGTH,TIME_MAX_LEN,
        0);

                      /* insert some spaces to align the read buttons */
    panel_create_item(met_panel, PANEL_MESSAGE,
        PANEL_LABEL_STRING,  "                          ",
        0);

   /* create read met button and register input_met with the notifier */
    panel_create_item(met_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(met_panel,
        "read  met", 0,0),
        PANEL_NOTIFY_PROC, input_met,
        0);

                          /* put top and bottom boundary on the panel */
    window_fit_height(met_panel);

}  /* end create_met_panel */
```

```
static Notify_value
mode_select(item, value, event)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: create_editing_mode_panel()

   PURPOSE: This routine establishes the mode that the editor is
            operating in by setting the global variable edit_mode to
            one of the predefined constants for the objects.

*************************************************************************/
Panel_item item;
int        value;
Event      *event;

{
    switch(value)
    {
        case OPERATOR:
                        edit_mode = OPERATOR;
                        break;

        case DATA_STREAM:
                        edit_mode = DATA_STREAM;
                        break;

        case SELF_LOOP:
                        edit_mode = SELF_LOOP;
                        break;

        case INPUT:
                        edit_mode = INPUT;
                        break;

        case OUTPUT:
                        edit_mode = OUTPUT;
                        break;

        default:
                        break;

    }
    return;

}    /* end mode select proc */
```

```
quit_proc()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: display_error_msg()

   CALLED BY: create_editing_mode_panel()

   PURPOSE: This routine checks to see if the diagram has been
            stored and if so, terminates the program and clears the
            display. If the diagram has not been saved it causes an
            error message to be displayed and does not terminate
            execution.

*************************************************************************/
{
    if (graph_saved)
    {
                                                /* destroy the window */
        window_set(frame, FRAME_NO_CONFIRM, TRUE, 0);
        window_destroy(frame);
    }
    else
    {
        display_error_msg(6);              /* graph not saved message */
    }

}  /* end quit_proc */
```

```
load_proc()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: create_op()
          append_to op_list()
          create_line()
          append_line_to_op()
          redraw_diagram()

   CALLED BY: create_operating_mode_panel()

   PURPOSE: This routine loads a diagram which is in the file
            graph.pic and causes it to be redrawn.

*************************************************************************/
{
    int  optype,x1,y1,x2,y2;
    Name *oname,
         *dest;
    Met  *tc;

    graph_saved = 0;
    g = fopen("graph.pic","r");

    while (!feof(g))
    {
        oname = (Name *)malloc(sizeof(Name));   /* alloc name storage */
        fscanf(g,"%d\n",&optype);                   /* read in values */
        fscanf(g,"%d\n",&x1);
        fscanf(g,"%d\n",&y1);
        fscanf(g,"%d\n",&x2);
        fscanf(g,"%d\n",&y2);
        fscanf(g,"%s\n",oname->string);
        oname->length = strlen(oname->string);   /* calc name length  */
        if ((optype == OPERATOR) || (optype == EXTERNAL))
        {
            tc = (Met *)malloc(sizeof(Met));    /* alloc met storage  */
            fscanf(g,"%s\n",tc->string);                 /* read met */
            tc->length = strlen(tc->string);       /* calc met length */
            op = create_op(oname,optype,x1,y1,x2,y2,tc); /*alloc node */
            append_to_op_list(op_list,op);     /* attach node to list */
        }
        else
        {
            dest = (Name *)malloc(sizeof(Name));/*alloc dest name stg */
            fscanf(g,"%s\n",dest->string);          /* read dest name */
            dest->length = strlen(dest->string);  /* calc name length */
```

```
            dop = create_op(dest,optype,x1,y1,x2,y2,tc); /*alloc node */
            ln = create_line(oname,optype,x1,y1,x2,y2,dop); /*alloc ln*/
            append_line_to_op(op,ln);     /* attach line to its source */
        }
    }
    fclose(g);
    redraw_diagram();

}  /* end load_proc */




store_proc()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: create_PSDL()
          store_diagram()

   CALLED BY: create_operating_mode_panel()

   PURPOSE: This routine stores the diagram in the file graph.pic.
            Prior to  storing the diagram it causes the picture to be
            transformed into the equivalent PSDL link statements.

*************************************************************************/
{
    create_PSDL();
    store_diagram();
    graph_saved = 1;

}  /* end store_proc */
```

```
static Notify_value
process_canvas_events(canvas,event)
/************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: pick_line()
          delete_line()
          pick_operator()
          delete_op()
          redraw_diagram()
          rubber_band()
          process_object()

   CALLED BY: main()

   PURPOSE: This routine processes canvas events. It determines what
            actions will take place for a given mouse event.

************************************************************************/
Canvas canvas;
Event  *event;
{
    int id = event_id(event);
    static int x1, y1, x2, y2;
    static int new_posit = 1;
    static int left_button = 0;
    static int middle_button = 0;
    Operator *op;
    Line *ln;

    if (event_is_button(event))            /* check for button events */
    {
        if (event_is_down(event))           /* check for down events */
        {
            x1 = event_x(event);     /* position of button down event */
            y1 = event_y(event);
            switch(id)
            {
                case MS_LEFT:                  /* save starting posit */
                    new_posit =1;
                    x2 = x1;
                    y2 = y1;
                    left_button=1;
                    break;

                case MS_MIDDLE:                        /* do nothing */
                    break;
```

```
                case MS_RIGHT:              /* pick object for deletion */
                    if ((ln = pick_line(op_list,x1,y1)) != NULL)
                    {
                        op = NULL;
                        delete_line(op_list,op,ln);
                    }
                    else if ((op= pick_operator(op_list,x1,y1)) != NULL)
                        delete_op(op_list,op);
                    redraw_diagram();
                    break;
            }
        }
        else if (event_is_up(event))            /* check for up events */
        {
            switch (id)
            {
                case MS_LEFT:
                    rubber_band(x1,y1,x2,y2);      /* blank out object */
                    x2 = event_x(event);      /* store new stop coords */
                    y2 = event_y(event);

                    process_object(x1,y1,x2,y2);     /* draw final obj */
                    left_button=0;
                    break;

                case MS_MIDDLE:                          /* do nothing */
                        break;

                case MS_RIGHT:                           /* do nothing */
                    break;
            }
        }
    }
    else if (id == LOC_DRAG)                  /* check for drag events */
    {
        if (left_button)
        {
            if (!new_posit)
                 /* rubber band operator's boundary while being drawn */
            {
                rubber_band(x1,y1,x2,y2);      /* erase previous image */
                x2 = event_x(event);            /* get new stop coords */
                y2 = event_y(event);
                rubber_band(x1,y1,x2,y2);         /* redraw the object */
            }
            else
                new_posit = 0;
        }
    }
    return;
}  /* end process_canvas_events */
```

```
process_object(x1,y1,x2,y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: pick_operator()
          process_operator()
          display_error_msg()
          process_line()

   CALLED BY: process_canvas_events()

   PURPOSE: This routine determines if an object meets the criteria
            to be drawn.

*************************************************************************/
int x1,y1,x2,y2;
{
    Operator *op, *sop, *dop;

    switch (edit_mode)
    {
        case OPERATOR:
            if (name_checked && met_checked)
            {                                   /* draw the object if it is*/
                                                /* not positioned on top of*/
                                                /* an existing object      */
                if  (((pick_operator(op_list,x1,y1))==NULL) &&
                     ((pick_operator(op_list,x2,y2))==NULL))
                {
                    process_operator(OPERATOR,x1,y1,x2,y2);
                }
            }
            else
            {
                display_error_msg(4);       /* name or met not checked */
            }
            break;

        case DATA_STREAM:
            if (name_checked)
            {
                                          /* draw line if it starts and */
                                           /* terminates on an operator */
                if (((sop=pick_operator(op_list,x1,y1))!=NULL) &&
                    ((dop=pick_operator(op_list,x2,y2))!=NULL) &&
                     (sop != dop))
                {
                    process_line(DATA_STREAM,x1,y1,x2,y2,sop,dop);
```

```
        }
    }
    else
    {
        display_error_msg(5);              /* name not checked */
    }
    break;

case SELF_LOOP:
    if (name_checked)
    {
                                    /* draw the loop if it      */
                                    /* starts on an object and  */
                                    /* is not intersecting      */
                                    /* an existing object       */
        if (((sop=pick_operator(op_list,x1,y1))!=NULL) &&
            ((dop=pick_operator(op_list,x2,y2))==NULL))
        {
            process_line(SELF_LOOP,x1,y1,x2,y2,sop,sop);
        }
    }
    else
    {
        display_error_msg(5);              /* name not checked */
    }
    break;

case INPUT:
    if (name_checked)
    {
                                        /* draw line if it ends */
                                            /* on an operator */
        if (((sop=pick_operator(op_list,x1,y1))==NULL) &&
            ((dop=pick_operator(op_list,x2,y2))!=NULL))
        {
            process_line(INPUT,x1,y1,x2,y2,sop,dop);
        }
    }
    else
    {
        display_error_msg(5);              /* name not checked */
    }
    break;

case OUTPUT:
    if (name_checked)
    {
        dop = NULL;
                                          /* draw the line if it */
                                        /* starts on an operator */
                                  /* and doesn't terminate on one */
```

```
                    if (((sop=pick_operator(op_list,x1,y1))!=NULL) &&
                        ((dop=pick_operator(op_list,x2,y2))==NULL))
                    {
                        process_line(OUTPUT,x1,y1,x2,y2,sop,dop);
                    }
                }
                else
                {
                    display_error_msg(5);              /* name not checked */
                }
                break;

            default:
                break;
        }

    }  /* end_process_object */
```

```
draw_object(otype,x1,y1,x2,y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_operator()
              process_line()
              redraw_diagram()

   PURPOSE: This routine causes the object to be drawn in final form.

*************************************************************************/
int  otype,x1,y1,x2,y2;

{
    float i,xmid,ymid,xcent,ycent;
    int xnew,ynew,xold,yold;


    switch(otype)
    {
        case OPERATOR:
                                              /* calc objects midpoint */
            xmid = (x2-x1)/2.0;
            ymid = (y2-y1)/2.0;
                                 /* objects center point on the screen */
            xcent = x1 + xmid;
            ycent = y1 + ymid;
                               /* position to start drawing the object */
            xold = x2;
            yold = ycent;

                                             /* loop to draw the object */
            for(i = 0.0; i <= 2 * PI; i = i + PI / 12)
            {
                                        /* draw line segments of an oval */
                xnew = xcent + (xmid * cos(i));
                ynew = ycent + (ymid * sin(i));
                pw_vector(drawing_pw, xold, yold, xnew, ynew, PIX_SRC,
                1);
                xold = xnew;
                yold = ynew;
            }
            break;
```

```
            case DATA_STREAM:
                                    /* draw line from (x1,y1) to (x2,y2) */
                pw_vector(drawing_pw, x1, y1, x2, y2, PIX_SRC, 1);
                break;

            case SELF_LOOP:
                                     /* draw three sides of a rectangle */
                pw_vector(drawing_pw, x2, y1, x2, y2, PIX_SRC, 1);
                pw_vector(drawing_pw, x2, y2, x1, y2, PIX_SRC, 1);
                pw_vector(drawing_pw, x1, y2, x1, y1, PIX_SRC, 1);
                break;

            case INPUT:
                                    /* draw line from (x1,y1) to (x2,y2) */
                pw_vector(drawing_pw, x1, y1, x2, y2, PIX_SRC, 1);
                break;

            case OUTPUT:
                                    /* draw line from (x1,y1) to (x2,y2) */
                pw_vector(drawing_pw, x1, y1, x2, y2, PIX_SRC, 1);
                break;

            default :
                break;
        }

    }  /* end draw_object */
```

```
rubber_band(x1,y1,x2,y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_canvas_events()

   PURPOSE: This routine draws the object during the rubber banding
            process. An operator is drawn as a rectangle by this
            routine.

*************************************************************************/
int  x1,y1,x2,y2;
{
    switch(edit_mode)
    {
        case OPERATOR:
            pw_vector(drawing_pw, x1, y1, x2, y1, PIX_NOT(PIX_DST), 1);
            pw_vector(drawing_pw, x2, y1, x2, y2, PIX_NOT(PIX_DST), 1);
            pw_vector(drawing_pw, x2, y2, x1, y2, PIX_NOT(PIX_DST), 1);
            pw_vector(drawing_pw, x1, y2, x1, y1, PIX_NOT(PIX_DST), 1);
            break;

        case DATA_STREAM:
            pw_vector(drawing_pw, x1, y1, x2, y2, PIX_NOT(PIX_DST), 1);
            break;

        case SELF_LOOP:
            pw_vector(drawing_pw, x2, y1, x2, y2, PIX_NOT(PIX_DST), 1);
            pw_vector(drawing_pw, x2, y2, x1, y2, PIX_NOT(PIX_DST), 1);
            pw_vector(drawing_pw, x1, y2, x1, y1, PIX_NOT(PIX_DST), 1);
            break;

        case INPUT:
            pw_vector(drawing_pw, x1, y1, x2, y2, PIX_NOT(PIX_DST), 1);
            break;

        case OUTPUT:
            pw_vector(drawing_pw, x1, y1, x2, y2, PIX_NOT(PIX_DST), 1);
            break;

        default:
            break;
    }
}  /* end rubber_band */
```

```
process_operator(otype,x1,y1,x2,y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: draw_object()
          get_met()
          display_name()
          display_met()
          create_op()
          append_to_op_list()

   CALLED BY: process_object()

   PURPOSE: This routine manages the process of drawing, labeling,
            storing,and linking an operator.

*************************************************************************/
int  otype,x1,y1,x2,y2;

{
    Name *obj_name;
    Met  *tc;
    Operator *op;


    draw_object(otype,x1,y1,x2,y2);
    tc = get_met();
    obj_name = get_name();
    display_name(obj_name,otype,x1,y1,x2,y2);
    display_met(tc,x1,y1,x2,y2);
    op = create_op(obj_name,otype,x1,y1,x2,y2,tc);
    append_to_op_list(op_list,op);
    name_checked = 0;
    met_checked = 0;

} /* end process_operator */
```

```
process_line(otype,x1,y1,x2,y2,sop,dop)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: draw_object()
          draw_arrowhead()
          get_met()
          external()
          create_op()
          append_to_op_list()
          get_name()
          display_name()
          create_line()
          append_line_to_op()

   CALLED BY: process_object()

   PURPOSE: This routine manages the process of drawing, labeling,
            storing, an linking lines.

*************************************************************************/
int otype,x1,y1,x2,y2;
Operator *sop,*dop;

{
    Name *obj_name,*op_name;
    Line *ln;

    draw_object(otype,x1,y1,x2,y2);
    if (otype == SELF_LOOP)
        draw_arrowhead(x2,y2,x2,y1);
    else
        draw_arrowhead(x1,y1,x2,y2);

    if (otype == INPUT)
    {
        tc = get_met();
        op_name = external();
        sop = create_op(op_name,EXTERNAL,x1,y1,x2,y2,tc);
        append_to_op_list(op_list,sop);
    }
    obj_name = get_name();
    display_name(obj_name,otype,x1,y1,x2,y2);
    ln = create_line(obj_name,otype,x1,y1,x2,y2,dop);
    append_line_to_op(sop,ln);
    name_checked = 0;

}  /* end process_line */
```

```
Operator *
create_op(name,op_type,x1,y1,x2,y2,tc)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: alloc_operator()

   CALLED BY: load_proc()
              process_operator()
              process_line()

   PURPOSE: This routine stores all the info for an operator in a
            structure.

*************************************************************************/
Name  *name;
int   op_type,x1,y1,x2,y2;
Met   *tc;
{
    Operator *new_op;
    new_op = alloc_operator();        /* get storage for the operator */
                                /* fill in the values for the operator */
    new_op->name   = name;
    new_op->met = tc;
    new_op->head = NULL;
    new_op->tail = NULL;
    new_op->next = NULL;
    switch(op_type)
    {
        case OPERATOR:
                    new_op->optype = OPERATOR;
                    new_op->xstart = x1;
                    new_op->ystart = y1;
                    new_op->xstop  = x2;
                    new_op->ystop  = y2;
                    break;
        case EXTERNAL:
                    new_op->optype = EXTERNAL;
                    new_op->xstart = 0;
                    new_op->ystart = 0;
                    new_op->xstop  = 0;
                    new_op->ystop  = 0;
                    break;
        default:
                    break;
    }
    return(new_op);
}  /* end create_op */
```

```
Line *
create_line(name,ln_type,x1,y1,x2,y2,dest_op)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: alloc_line()

   CALLED BY: load_proc()
              process_line()

   PURPOSE: This routine stores the information for a line in a line
            structure.

*************************************************************************/
Name     *name;
int      ln_type;
int      x1,y1,x2,y2;
Operator *dest_op;
{
    Line *new_ln;
                                          /* get storage for the line */
    new_ln = alloc_line();
                                       /* fill in the values for the line */
    new_ln->name   = name;
    new_ln->xstart = x1;
    new_ln->ystart = y1;
    new_ln->xstop  = x2;
    new_ln->ystop  = y2;
    new_ln->next   = NULL;
    switch(ln_type)
    {
        case INPUT:
                     new_ln->lntype = INPUT;
                     new_ln->dest   = dest_op->name;
                     break;

        case DATA_STREAM:
                     new_ln->lntype = DATA_STREAM;
                     new_ln->dest   = dest_op->name;
                     break;

        case OUTPUT:
                     new_ln->lntype = OUTPUT;
                     new_ln->dest   = external();
                     break;

        case SELF_LOOP:
                     new_ln->lntype = SELF_LOOP;
```

```
                        new_ln->dest    = dest_op->name;
                        break;

            default:
                        break;
        }
        return(new_ln);

    }  /* end create_line */



delete_op(op_list,op)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: delete_input_lines()

   CALLED BY: process_canvas_events()

   PURPOSE: This routine deletes an operator from the list of
            operators and frees up the storage.

*************************************************************************/
Operator_list *op_list;
Operator *op;

{
    Operator *dptr,
             *otemp;
    Line     *lptr,
             *ltemp;
    Name     *n;

                                    /* find lines which terminate on */
                                    /* this operator and delete them */
    n = op->name;
    delete_input_lines(op_list,n);

    otemp = op_list->head;          /* put pointer at head of op list */

    if (op != otemp)            /* is the first op the one to delete? */
    {
        while (otemp->next != op)   /* if not, find the one to delete */
        {
            otemp = otemp->next;
        }
        dptr = otemp->next;          /* unlink the op to be deleted   */
```

```
            if (dptr->next != NULL)
            {
                otemp->next = dptr->next;
                dptr->next = NULL;
            }
            else
                otemp->next = NULL;
        }
        else                        /* the first one is the one to delete */
        {
            dptr = op_list->head;                   /* unlink the first op */
            op_list->head = dptr->next;
        }
        if (dptr->head != NULL)          /* does it have any assoc lines ? */
        {
            ltemp = dptr->head;            /* if so, free up their storage */
            lptr = dptr->head;
            dptr->head = NULL;
            while(lptr->next != NULL)
            {
                lptr = lptr->next;
                free(ltemp);
                ltemp = lptr;
            }
            free(lptr);                 /* free up the last line's storage */
            lptr = NULL;
            ltemp = NULL;
        }
        free(dptr);                      /* free up the operator's storage */
        dptr = NULL;

    }/* end delete_op */
```

```
delete_input_lines(op_list,n)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: delete_line()

   CALLED BY: delete_op()

   PURPOSE: This routine finds all of the input lines for an
            operator which has been deleted and causes them to be
            deleted.

*************************************************************************/
Operator_list *op_list;
Name *n;

{
    Operator *optr;
    Line     *lptr,
             *ltemp;

    optr = op_list->head;           /* start at head of operator list */
    while(optr != NULL)        /* search the entire list of operators */
    {
        lptr = optr->head;     /* start inner ptr at head of line list */
        while(lptr != NULL)  /* check each line leaving each operator */
        {
            if(!strcmp(n->string,lptr->dest->string))
            {
                ltemp = lptr->next;
                                              /* found a line with the */
                                            /* destination searched for */
                delete_line(op_list,optr,lptr);       /* so delete it */
                lptr = ltemp;
            }
            else
            {
                lptr = lptr->next;             /* go to the next line */
            }
        }
        optr = optr->next;                 /* go to the next operator */
    }

} /* end delete_input_lines */
```

```
delete_line(op_list,op,ln)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: delete_input_lines()
              process_canvas_events()

   PURPOSE: This routine deletes a line from an operator's line
            list. If the source operator is not known, it will search
            through all of thelists until it is found.

*************************************************************************/
Operator_list *op_list;
Operator      *op;
Line          *ln;

{
    Operator *optr;
    Line     *lptr,
             *ltemp;
    int      ln_found = 0;

    if(op != NULL)                    /* the line's source op is known */
    {
        optr = op;  /* start the search for the line at its source op */
        lptr = optr->head;
        ltemp = lptr;
        while(lptr != ln)          /* skip the lines which don't match */
        {
            ltemp = lptr;
            lptr = lptr->next;
        }
    }
    else                         /* source op is unknown - find the line */
    {
        optr = op_list->head;         /* start at beginning of op list */
        while((optr != NULL) && (!ln_found))     /* search until found */
        {
            lptr = optr->head;           /* start at head of line list */
            ltemp = lptr;
            while((lptr != NULL) && (!ln_found))   /*search line list */
            {
                if (lptr == ln)                     /* found the line */
                {
                    ln_found = 1;
                }
```

```
                    else
                    {
                        ltemp = lptr;                   /* skip to next line */
                        lptr = lptr->next;
                    }
                }

                if (!ln_found)
                    optr = optr->next; /*search next operator's line list */
            }
        }
                                /* unlink the line and free its storage */
            if (ltemp == lptr)                  /* delete first line on list */
            {
                optr->head = ltemp->next;
                lptr->next = NULL;
                ltemp = NULL;
                if (optr->head == NULL)    /* first line was the only line */
                {
                    optr->tail = NULL;
                }
            }
            else if(lptr == optr->tail) /* delete last line from the list */
            {
                ltemp->next = NULL;
                optr->tail = ltemp;
            }
            else                        /* delete a middle line from the list */
            {
                ltemp->next = lptr->next;
                lptr->next = NULL;
            }
            free(lptr);
            optr = NULL;

    } /* end delete_line */
```

```
Operator *
pick_operator(op_list,xpick,ypick)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: is_op_pick()

   CALLED BY: process_canvas_events()
              process_object()

   PURPOSE: This routine determines if a data stream or output
            line starts on an operator. If so, it returns a pointer to
            that operator.

*************************************************************************/
Operator_list *op_list;
int           xpick, ypick;

{
    Operator *ptr;

                                                /* search operator list */
    for (ptr = op_list->head; ptr != NULL; ptr = ptr->next)
    {
        if (ptr->optype == OPERATOR)        /* skip the null operators */
        {
                                                    /* test for pick */
            if (is_op_pick(ptr->xstart, ptr->ystart,
                           ptr->xstop, ptr->ystop, xpick, ypick))
            {
                return(ptr);
            }
        }
    }
    return(NULL);

}  /* end pick_operator */
```

```
int is_op_pick(x1,y1,x2,y2,xp,yp)
/*****************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: pick_operator()

   PURPOSE: This routine determines if a pick has occurred within
            the bounds of an operator.

*****************************************************************************/
int   x1, y1, x2, y2, xp, yp;

{
    if (( (xp > x1) && (xp < x2) && (yp > y1) && (yp < y2) )||
        ( (xp < x1) && (xp > x2) && (yp > y1) && (yp < y2) )||
        ( (xp < x1) && (xp > x2) && (yp < y1) && (yp > y2) )||
        ( (xp > x1) && (xp < x2) && (yp < y1) && (yp > y2) ))

        return(1);
    else
        return(0);

}  /* end is_op_pick */
```

```
Line *
pick_line(op_list,xpick,ypick)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: is_line_pick()

   CALLED BY: process_canvas_events()

   PURPOSE: This routine scans each operator's line list checking
            for a line which has been picked.

*************************************************************************/
Operator_list *op_list;
int           xpick, ypick;

{
    Operator *optr;
    Line     *lptr;
                                          /* scan the operator list */
    for (optr = op_list->head; optr != NULL; optr = optr->next)
    {
                                            /* scan each line list */
        for (lptr = optr->head; lptr != NULL; lptr = lptr->next)
        {
                                                  /* test for pick */
            if (is_line_pick(lptr->xstart, lptr->ystart,
                             lptr->xstop, lptr->ystop, xpick, ypick))
            {
                return(lptr);
            }
        }
    }
    return(NULL);

}  /* end pick_line */
```

```
int is_line_pick(x1,y1,x2,y2,xp,yp)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: pick_line()

   PURPOSE: This routine checks to see if a pick has occurred
            within the pick region of a line.

*************************************************************************/
int   x1, y1, x2, y2, xp, yp;

{
                          /* is (xp,yp) within pick region of a line? */
    if (((abs(x1-xp)+abs(y1-yp)) < PROXIMITY) ||
        ((abs(x2-xp)+abs(y2-yp)) < PROXIMITY))

        return(1);
    else
        return(0);

}  /* end is_line_pick */
```

```
append_to_op_list(op_list,op)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: load_proc()
              process_operator()
              process_line()

   PURPOSE: This routine attaches a newly created operator to the
            list of operators.

*************************************************************************/
Operator_list *op_list;
Operator      *op;

{
    if (op_list->head == NULL)
    {
        op_list->head = op;          /* attach first operator to list */
        op_list->tail = op;
    }
    else
    {
        op_list->tail->next = op;    /* attach operator to end of list */
        op_list->tail = op;
    }

}  /* end append_to_op_list */
```

```
append_line_to_op(op,ln)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: load_proc()
              process_line()

   PURPOSE: This routine attaches lines which depart an operator to that
            operator's list of output lines.

*************************************************************************/
Operator *op;
Line     *ln;

{
    if (op->head == NULL)
    {
        op->head = ln;                    /* attach first line to list */
        op->tail = ln;
    }
    else
    {
        op->tail->next = ln;             /* attach to end of line list */
        op->tail = ln;
    }

}  /* end append_line_to_op */
```

```
Operator *
alloc_operator()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: create_op()

   PURPOSE: This routine allocates the storage for an operator
            structure.

*************************************************************************/
{
    Operator *op;

    op = (Operator *)malloc(sizeof(Operator));
    return(op);

} /* end alloc_operator */




Line *
alloc_line()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: create_line()

   PURPOSE: This routine allocates the storage for a line structure.

*************************************************************************/
{
    Line *ln;

    ln = (Line *)malloc(sizeof(Line));
    return(ln);

} /* end alloc_line */
```

```
input_name(item, value, event)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: is_valid_ada_id()
          display_error_msg()

   CALLED BY: create_name_panel()

   PURPOSE: This routine reads the names from the name panel and
            checks to see if they are valid ada identifier names. If
            they are invalid,an error message is displayed.

*************************************************************************/
Panel_item    item;
int           value;
Event        *event;
{
                                          /* get storage for the string */
    name_buf = malloc(40);
                                              /* initialize the storage */
    name_buf[0] = '\0';

                              /* read the users string from the display */
    strcpy(name_buf, (char *)panel_get_value(object_name));

                     /* check to see if the name is an ada identifier */
    if (is_valid_ada_id(name_buf))
    {
        display_error_msg(1);      /* blank out the error message area */
        name_checked = 1;
    }
    else
    {
        display_error_msg(2);                 /* invalid ada id message */
    }

}  /* end input_name */
```

```
display_error_msg(msg_id)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: quit_proc()
              process_object()
              input_name()
              input_met()

   PURPOSE: This routine displays error messages on the canvas
            in response to checks made throughout the system. A numeric
            code signifies which message to display.

*************************************************************************/
int   msg_id;
{
    char  *message;

    message = malloc(45);
    switch(msg_id)
    {
        case 1:
                message = "                                              ";
                break;
        case 2:
                message = "SYNTAX ERROR in ADA identifier                ";
                break;
        case 3:
                message = "SYNTAX ERROR in Maximum Execution Time        ";
                break;
        case 4:
                message = "ERROR - Either NAME or MET not read           ";
                break;
        case 5:
                message = "ERROR - NAME not read                         ";
                break;
        case 6:
                message = "WARNING - The graph has not been stored!      ";
                break;
        default:
                message = "";
                break;
    }
    pw_text(canvas_pixwin(drawing_canvas),150,15,PIX_SRC,NULL,message);

}  /* end display_error_msg */
```

```
Name *
get_name()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_operator()
              process_line()

   PURPOSE: This routine allocates the storage required for
            a name, stores the name in the structure, and returns a
            pointer to it.

*************************************************************************/
{
    Name *n;

                                          /* get storage for the name */
    n = (Name *)malloc(sizeof(Name));
                                      /* assign the name to this storage */
    strcpy(n->string,name_buf);
                                            /* find the name's length */
    n->length = strlen(n->string);
    return(n);

}  /* end get_name */
```

```
display_name(n,otype,x1,y1,x2,y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_operator()
              process_line()
              redraw_diagram()

   PURPOSE: This routine displays the name of an object which
            the user has drawn.

*************************************************************************/
Name  *n;
int   otype,x1,y1,x2,y2;

{
    float  xcent,ycent;

                                          /* calc center of the object */
    xcent = x1 + ((x2 - x1) / 2.0);
    ycent = y1 + ((y2 - y1) / 2.0);

    switch(otype)
    {
        case OPERATOR:
                    pw_text(canvas_pixwin(drawing_canvas),
                            (int)xcent-((n->length)/2)*8,
                            (int)ycent+5,PIX_SRC,NULL,n->string);
                    break;

        case INPUT:
                    pw_text(canvas_pixwin(drawing_canvas),x1,y1,
                            PIX_SRC, NULL,n->string);
                    break;

        case OUTPUT:
                    pw_text(canvas_pixwin(drawing_canvas),x2,y2,
                            PIX_SRC, NULL,n->string);
                    break;

        case DATA_STREAM:
                    xcent = (x2 - x1) / 2.0;
                    ycent = (y2 - y1) / 2.0;
                    pw_text(canvas_pixwin(drawing_canvas),
                            x1+(int)xcent,y1+(int)ycent,PIX_SRC,
                            NULL,n->string);
```

```
                        break;

            case SELF_LOOP:
                        pw_text(canvas_pixwin(drawing_canvas),
                                (int)xcent-((n->length)/2)*8,
                                (int)y2-7,PIX_SRC,NULL,n->string);
                        break;
            default:

                        break;
        }

    }  /* end display_name */
```

```
int
is_valid_ada_id(name_buf)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: input_name()

   PURPOSE: This routine checks to see if a name is a legal ada
            identifier

*************************************************************************/
char  name_buf[40];

{
    int    space_found = 0;
    int    i = 1;

    if (isalpha(name_buf[0]))
    {
        while (i <= strlen((char *)name_buf) - 1)
        {
            if(!isgraph(name_buf[i]))
            {
                space_found = 1;
                i = i + 1;
            }
            else if(((isalnum(name_buf[i]))||
                     (name_buf[i]=='_'))&& !(space_found))
            {
                i = i + 1;
            }
            else
            {
                return(0);
            }
        }
        return(1);
    }
    else
    {
        return(0);
    }

}  /* end is_valid_ada_id */
```

```
input_met(item, value, event)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: is_valid_met()
          display_error_msg()

   CALLED BY: create_met_panel()

   PURPOSE: This routine reads a MET from the met panel and checks
            to see if it is valid. If it is not, an error message
            is generated.

*************************************************************************/
Panel_item    item;
int           value;
Event        *event;
{
                                        /* get storage for the met value */
    met_buf = malloc(12);
                                                        /* initialize */
    met_buf[0] = '\0';

                                                /* read in the met string */
    strcpy(met_buf, (char *)panel_get_value(time_constraint));
    if (is_valid_met(met_buf))
    {
        display_error_msg(1);           /* blank out the message area */
        met_checked = 1;
    }
    else
    {
        display_error_msg(3);                           /* invalid met */
    }

}  /* end input_met */
```

```
int
is_valid_met(met_buf)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: is_valid_met()

   PURPOSE: This routine checks an MET to see if it has the proper
            syntax.

*************************************************************************/
char  met_buf[12];

{
    int   nondigit_found = 0;
    int   letter_prev_found = 0;
    int   mfound = 0;
    int   ufound = 0;
    int   done =0;
    int   i = 1;

    if (isdigit(met_buf[0]))
    {
        while (i <= strlen((char *)met_buf) - 1)
        {
            if (!isalnum(met_buf[i]))
            {
                return(0);
            }
            else if (isdigit(met_buf[i])&&!nondigit_found)
            {
                i = i + 1;
            }
            else if (isdigit(met_buf[i]))
            {
                return(0);
            }
            else
            {
                nondigit_found = 1;
                switch(met_buf[i])
                {
                    case 'u': if (!letter_prev_found)
                              {
                                  ufound = 1;
                                  letter_prev_found = 1;
```

```
                                    i = i + 1;
                                }
                                else
                                {
                                    return(0);
                                }
                                break;

                        case 's': if (!letter_prev_found)
                                {
                                    letter_prev_found = 1;
                                    i = i + 1;
                                }
                                else if ((ufound||mfound)&&!done)
                                {
                                    done = 1;
                                    i = i + 1;
                                }
                                else
                                {
                                    return(0);
                                }
                                break;

                        case 'm': if (!letter_prev_found)
                                {
                                    mfound = 1;
                                    letter_prev_found = 1;
                                    i = i + 1;
                                }
                                else
                                {
                                    return(0);
                                }
                                break;

                        default : return(0);
                                break;
                    }
                }
            }
            if ((ufound&&!done)||(!nondigit_found))
                return(0);
        }
        else
        {
            return(0);
        }
    } /* end is_valid_met */
```

```
Met *
get_met()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_operator()
              process_line()

   PURPOSE: This routine allocates a storage structure for a
            MET and fills it in with the MET which has been entered.
            It then returns a  pointer to this structure.


*************************************************************************/
{
    Met *tc;


                                          /* get storage for the met */
    tc = (Met *)malloc(sizeof(Met));

    switch(edit_mode)
    {
        case OPERATOR:
                              /* assign the input string to this storage */
                    strcpy(tc->string,met_buf);
                                  /* find the length of the string */
                    tc->length = strlen(tc->string);
                    break;

        case INPUT:
                    strcpy(tc->string,"0s");
                    tc->length = 2;
                    break;

        default:
                    break;
    }
    return(tc);

}  /* end get_met */
```

```
display_met(tc,x1,y1,x2,y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_operator()
              redraw_diagram()

   PURPOSE: This routine displays an operator's MET in a position
            centered directly above it.

*************************************************************************/
Met *tc;
int x1,y1,x2,y2;

{
    float  xcent;


    xcent = x1 + ((x2 - x1) / 2.0);
    pw_text(canvas_pixwin(drawing_canvas),
           (int)xcent-((tc->length)/2)*8,
           (int)y1-5,PIX_SRC,NULL,tc->string);

}  /* end display_met */
```

```
Name *
external()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_line()

   PURPOSE: This routine returns the name "EXTERNAL whenever
            it is called. EXTERNAL is the name of the source operator
            for all input lines and the destination of all output
            lines.

*************************************************************************/
{
    Name *n;

    n = (Name *)malloc(sizeof(Name));              /* alloc storage */
    strcpy(n->string,"EXTERNAL");                    /* assign name */
    n->length = 8;                                   /* assign name */
    return(n);

}  /* end external */
```

```
create_PSDL()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: store_proc()

   PURPOSE: This routine creates the PSDL statements represented
            by the user's data flow diagram. A PSDL statement of the
            form

        output_line_name.source_name[:met]-->destination_name

            will be constructed from the information contained in the
            operator list.

*************************************************************************/
{
    Operator *op_ptr;
    Line     *output_ptr;
    char     *psdl;

    char     *period = ".";
    char     *colon  = ":";
    char     *arrow  = "-->";

    f = fopen("graph.link","w");
    psdl = malloc(150);
    op_ptr = op_list->head;     /* point at head of the operator list */
    while(op_ptr != NULL)
    {
        output_ptr = op_ptr->head; /* pt line ptr at head of line list*/
        while(output_ptr != NULL)
        {
                                   /* assemble the psdl statement by */
                                   /* concatenating the parts of the */
                                        /*  PSDL statements together */
            psdl = strcat(psdl,output_ptr->name->string);
            psdl = strcat(psdl,period);
            psdl = strcat(psdl,op_ptr->name->string);
            if ((op_ptr->optype == OPERATOR) &&  /* put in met for op */
                (op_ptr->met->string != "0s"))
            {
                psdl = strcat(psdl,colon);
                psdl = strcat(psdl,op_ptr->met->string);
            }
            psdl = strcat(psdl,arrow);
```

```
            psdl = strcat(psdl,output_ptr->dest->string);
            fprintf(f,"%s\n",psdl);          /* store link stmt in file */
            psdl[0] = '\0';                          /* reinitialize */
            output_ptr = output_ptr->next;
        }
        op_ptr = op_ptr->next;
    }
    fclose(f);

}  /*  end create_PSDL  */
```

```
store_diagram()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: store_proc()

   PURPOSE: This routine stores the contents of the entire
            storage structure for the diagram in a file named
            graph.pic. It writes out each  operator followed
            immediately by the lines which leave it.

*************************************************************************/
{
    Operator *op_ptr;
    Line     *ln_ptr;

    g = fopen("graph.pic","w");
    op_ptr = op_list->head;                    /* start at head of list */
    while(op_ptr != NULL)
    {
        fprintf(g,"%d\n",op_ptr->optype);          /* output each node */
        fprintf(g,"%d\n",op_ptr->xstart);
        fprintf(g,"%d\n",op_ptr->ystart);
        fprintf(g,"%d\n",op_ptr->xstop);
        fprintf(g,"%d\n",op_ptr->ystop);
        fprintf(g,"%s\n",op_ptr->name->string);
        fprintf(g,"%s\n",op_ptr->met->string);

        ln_ptr = op_ptr->head;          /* start at head of operator's */
                                                          /* line list */
        while(ln_ptr != NULL)
        {                                  /* output line node contents */
            fprintf(g,"%d\n",ln_ptr->lntype);
            fprintf(g,"%d\n",ln_ptr->xstart);
            fprintf(g,"%d\n",ln_ptr->ystart);
            fprintf(g,"%d\n",ln_ptr->xstop);
            fprintf(g,"%d\n",ln_ptr->ystop);
            fprintf(g,"%s\n",ln_ptr->name->string);
            fprintf(g,"%s\n",ln_ptr->dest->string);
            ln_ptr = ln_ptr->next;             /* advance to next line */
        }
        op_ptr = op_ptr->next;             /* advance to next operator */
    }
    fclose(g);

}  /* end store_diagram */
```

```
redraw_diagram()
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: draw_object()
          display_name()
          display_met()
          draw_arrowhead()

   CALLED BY: load_proc()
              process_canvas_events()

   PURPOSE: This routine blanks out the screen and redraws each object
            stored in the operator list and each operator's line list.

*************************************************************************/
{
    Operator *op_ptr;
    Line     *ln_ptr;

                                                  /* blank the screen */
    pw_writebackground(drawing_pw,0,0,
                       window_get(drawing_canvas,CANVAS_WIDTH),
                       window_get(drawing_canvas,CANVAS_HEIGHT),
                       PIX_SRC);
    op_ptr = op_list->head;                  /* start at head of list */

    while(op_ptr != NULL)
    {
        if(op_ptr->optype == OPERATOR)
        {                                          /* draw the object */
            draw_object(op_ptr->optype,op_ptr->xstart,op_ptr->ystart,
                        op_ptr->xstop,op_ptr->ystop);
            display_name(op_ptr->name,OPERATOR,op_ptr->xstart,
                         op_ptr->ystart,op_ptr->xstop,op_ptr->ystop);
            display_met(op_ptr->met,op_ptr->xstart,op_ptr->ystart,
                        op_ptr->xstop,op_ptr->ystop);
        }
        ln_ptr = op_ptr->head;          /* start at head of line list */

        while(ln_ptr != NULL)
        {                                           /* draw the line */
            draw_object(ln_ptr->lntype,ln_ptr->xstart,ln_ptr->ystart,
                        ln_ptr->xstop,ln_ptr->ystop);
                                                /* attach its arrowhead */
            if (ln_ptr->lntype == SELF_LOOP)
                draw_arrowhead(ln_ptr->xstop,ln_ptr->ystop,
                               ln_ptr->xstop,ln_ptr->ystart);
```

```
                else
                    draw_arrowhead(ln_ptr->xstart,ln_ptr->ystart,
                                   ln_ptr->xstop,ln_ptr->ystop);

                display_name(ln_ptr->name,ln_ptr->lntype,ln_ptr->xstart,
                             ln_ptr->ystart,ln_ptr->xstop,ln_ptr->ystop);

                ln_ptr = ln_ptr->next;                  /* adv line pointer */
            }
            op_ptr = op_ptr->next;          /* advance the operator pointer */
        }

    } /* end redraw diagram */
```

```
draw_arrowhead(x1, y1, x2, y2)
/*************************************************************************

   WRITTEN BY: R. K. Thorstenson

   LAST MODIFIED: 10 Nov 1988

   CALLS: N/A

   CALLED BY: process_line()
              redraw_diagram()

   PURPOSE: This routine draws an arrowhead at the end of a line at the
            appropriate angle.

*************************************************************************/
int x1, y1, x2, y2;
{
    int    x1_trans, y1_trans, x2_trans, y2_trans;
    int    xpt1, ypt1, xpt2, ypt2,
           xpt1_trans, ypt1_trans, xpt2_trans, ypt2_trans;
    double length,
           theta;

                                    /* translate the line to the origin */
    x1_trans = x1 - x2;
    y1_trans = y1 - y2;

                                        /* find the length of the line */
    length = sqrt(pow((double)x1_trans,2.0) +
                  pow((double)y1_trans,2.0));

                    /* find the angle between the line and the x axis */
    theta = acos ((double)x1_trans/length);

              /* calculate the coords of the points of the arrowhead */
    xpt1 = ARROW_LENGTH * cos(theta + PI / 6.0);
    ypt1 = ARROW_LENGTH * sin(theta + PI / 6.0);

    xpt2 = ARROW_LENGTH * cos(theta - PI / 6.0);
    ypt2 = ARROW_LENGTH * sin(theta - PI / 6.0);

           /* reflect y coords across x axis if y1_trans is negative */
    if (y1_trans < 0)
    {
        ypt1 = -ypt1;
        ypt2 = -ypt2;
    }

     /* translate the arrowhead's coords out to the posit of the line */
    xpt1_trans = xpt1 + x2;
```

```
        ypt1_trans = ypt1 + y2;
        xpt2_trans = xpt2 + x2;
        ypt2_trans = ypt2 + y2;

                                            /* draw the point of the arrow */
        pw_vector(drawing_pw, xpt1_trans, ypt1_trans, x2, y2, PIX_SRC, 1);
        pw_vector(drawing_pw, xpt2_trans, ypt2_trans, x2, y2, PIX_SRC, 1);
        pw_vector(drawing_pw, xpt1_trans, ypt1_trans, xpt2_trans,
                  ypt2_trans, PIX_SRC, 1);

    } /* end draw arrow head */
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Professor Luqi Code 52LQ Naval Postgraduate School Computer Science Department Monterey, California 93943-5000	1
4.	LT Roger K. Thorstenson 594-B Michelson Road Monterey, California 93940	1
5.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, Virginia 22217-5000	1
6.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, D.C. 20363-5100	1
7.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, D.C. 20301	1
8.	Naval Sea Systems Command Attn. CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 22202	1
9.	Office of the Secretary of Defense Attn. CDR Barber STARS Program Office Washington, D.C. 20301	1
10.	Office of the Secretary of Defense Attn. Mr. Joel Trimble STARS Program Office Washington, D.C. 20301	1

11. Commanding Officer 1
Naval Research Laboratory
Code 5150
Attn. Dr. Elizabeth Wald
Washington, D.C. 20375-5000

12. Navy Ocean System Center 1
Attn. Linwood Sutton, Code 423
San Diego, California 92152-500

13. National Science Foundation 1
Attn. Dr. William Wulf
Washington, D.C. 20550

14. National Science Foundation 1
Division of Computer and Computation Research
Attn. Dr. Peter Freeman
Washington, D.C. 20550

15. National Science Foundation 1
Director, PYI Program
Attn. Dr. C. Tan
Washington, D.C. 20550

16. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy Street
Arlington, Virginia 22217-5000

17. Office of Naval Research 1
Applied Mathematics and Computer Science, Code 1211
Attn. Mr. J. Smith
800 N. Quincy Street
Arlington, Virginia 22217-5000

18. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Jacob Schwartz
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

19. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Squires
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

20. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

21. Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

22. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

23. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

24. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

25. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

26. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

27. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, New York 13441-5700

28. Professor Michael Zyda 1
Code 52ZK
Naval Postgraduate School
Computer Science Department
Monterey, California 93943-5000